



Advanced Network Programming (ANP) Course Project Handbook

(XB_0048)

P2 (November - December), 2022

Version: 1.3

Animesh Trivedi (a.trivedi@vu.nl) and Lin Wang (lin.wang@vu.nl)

Malek Kanaan, Peter-Jan Gootzen, and Alessandro Balducci

Advanced Network Programming (ANP) Course Project Handbook	0
Revision Information	2
0. Code of Conduct (Plagiarism)	2
1. Overview	3
2. Logistics	3
2.1 What is given to you	4
2.2 How to ask for help when stuck in the project	5
2.3 Late/Slip days	6
2.4 Group Management Logistics	6
3. Assessment Criteria	7
3.1 Grading guidelines	7
3.2 Specific notes on partial assessment for Milestones	9
4. Background Network Reading Material	10
5. Project ANP Netstack: Building my own networking stack!	11
4.1 Milestone 1: Welcome to the machine	12
4.2 Milestone 2 : Is anyone out there?	23
4.3 Milestone 3: Hey you	25
4.4 Milestone 4: Careful With That Data, Eugene	35
4.5 Milestone 5: Another Graph in the Wall	37
5. Bonus Ideas and Timeline	38
Appendix	39
A. Important RFCs	39

Revision Information

- October 28th, 2022 : version 1.3 released for the academic year 2022
- September 1st, 2021: version 1.2 released for the academic year 2021

0. Code of Conduct (Plagiarism)

We have a ZERO-TOLERANCE cheating, fraud, and plagiarism policy.

You are encouraged to discuss specific issues with other students, search online, read Linux/Storage code, and share knowledge with other fellow students. After all, the whole field of computer science is built with open-collaboration. ***HOWEVER, in any circumstances you are expected to write your own code. Do not buy or sell code online. Do not copy code. Do not give out code to other students.*** It is quite unlikely that a project of this scope will result in two students/groups writing almost identical code. Any similarities in the code, or style will lead to further investigation, with a possibility of reporting to the examination board - an outcome we both would like to avoid.

1. Overview

The objective of this course assignment is to develop a working networking stack in user space. With this networking stack, you will be able to send and receive packets, process them, and transfer data for an application. Furthermore, you will learn about how to do empirical benchmarking, collect data, plot data points, and explain what is happening in your measurements.

More specifically you will need to implement:

1. **ICMP protocol** over IP to get the ping command working
2. **TCP protocol** socket with networking systems calls (socket, connect, send, recv, close). Your network protocol implementation will be able to run unmodified Linux networking programs (any, even firefox! - that is the bonus).

The aim of this project is to give an impression of the sophistication, complexity, and intricate nature of a networking stack implementation which is intertwined with scheduling, memory management, scalability, and performance-related concerns inside an operating system. Furthermore, as Moore's law is coming to an end, userspace specialization of networking stacks is one of the most promising and popular ways of delivering performance to applications.

2. Logistics

There are **5 milestones** in the project. The project is a team project, but we first start with individual milestones. This way you can assess your and your teammate's capabilities before making teams. Milestones 1 and 2 are individual assignments, and then from week 3 (including milestone 3) onwards all of them are group assignments. **You will work in a group of maximum of 3 members (fewer are, of course, allowed).**

Expectation management - what is in the lab and practical project

Unlike other courses that you might have taken, in this course you do not implement what we cover in the class. The expectation is that with your background knowledge, details covered in the lecture, and the given framework you are ready to implement a fully functional networking stack. The aim of the course is to make you familiar with the complexity, state-of-the-practice networking tools, and behind the scene work. Hence, there are no weekly lab tutorials, but more like doubts and clarification sessions. *Much of the work done will be done by your outside the lab hours. Keep this in mind, and try to make use of the lab sessions to get as much information as possible by preparing your questions and issues beforehand.* Many ideas that we cover in the lectures can be attempted as a bonus for the project. In the past edition of this course, multiple groups did that.

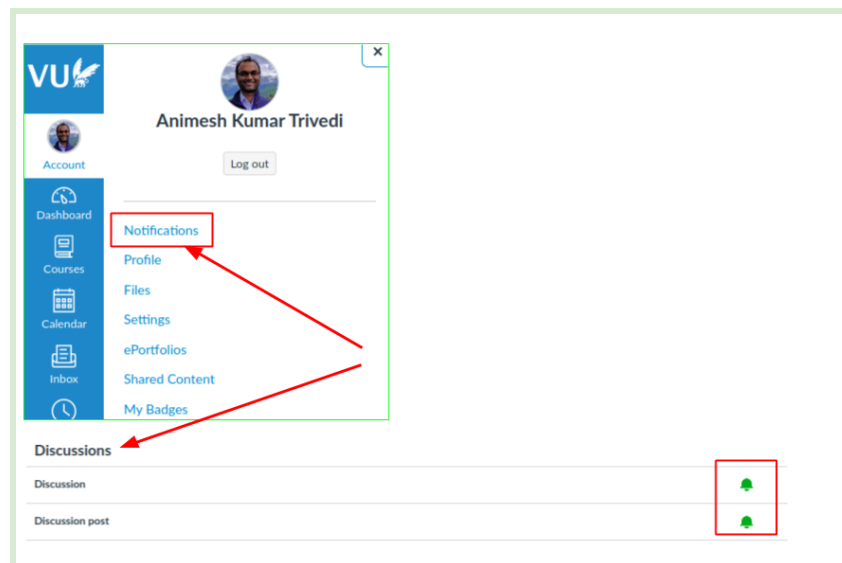
When are the deadlines: Please check the Canvas page, and ensure that you fulfill all the evaluation criteria. The whole framework and this complete handbook are released to you on day 1 of the course, and feel free to code ahead and finish as much as possible. On the demo day, you can still just demo us the required functionality while being ahead in the project time. You can use the additional time for bonuses. See the bonus section later in this handbook.

Weekly lab hours: There are weekly lab sessions. Please check the Canvas / Rooster. The lab hours are open to any sort of discussion including but not limited to questions about the project (we assume that would be the majority of the questions), clarification about lecture topics, general questions about research, etc.

Some pertinent and practical issues to consider:

- Be aware that you need to form a group in the second week. If you do not know anyone or have not managed to do so, then post it on the Discussion board of the Canvas page. Ask around actively and ask us if you are having difficulty forming groups.
- You can choose to do the milestones alone, but we do not recommend that. You will not get any lenient marking from us just because you choose to do it alone.
- In case you did not form a group by week 2 and did not communicate with us when milestone 3 starts we will assume that you are not planning to continue the course.

Important: Please enable notifications (<https://canvas.vu.nl/profile/communication>, you can also do course-specific settings) for the Discussion forum as we will be posting many clarifications and information there. In any case, do not forget to repeatedly check the information there. See this image to enable notifications:



2.1 What is given to you

We have provided a basic netskeleton framework in which you will develop the project. This framework is a CMake project with automated checks integrated for various milestones. Please make yourself

familiar with CMake/C/C++ (and if needed gdb, and friends). We assume that you are familiar with the basic executable concepts in Linux (shared library, compilation, running and installing packages and programs from the source). Please attend the first lab session. The first milestone is about building and deploying the netskeleton shared library and executables.

We also strongly recommend using a modern C/C++ IDE development environment like CLion or VSCode (with the CMake and C/C++ plugins).

Within the netskeleton framework, you are also given a simple TCP socket server/client program (in the server-client folder). This program works on the Linux kernel stack implemented in the kernel. At the end of this project, this program should work (unmodified) on your network stack too! And if you are adventurous, you can use this framework to run your browser on it too, see the Bonuses.

Note: we do not have any explicit support to do this project on Windows or Windows' Linux subsystem. So please make sure that you have a working setup in the first week.

2.2 How to ask for help when stuck in the project

Start early and identify the area where you need help. The lab sessions are exactly for that purpose. We cannot guarantee any help hours before the deadline, late in the evenings, or on weekends. So, please plan ahead, and ask with plenty of due time ahead.

So here is the protocol (pun intended, after all, we will be talking about protocols a lot in this course) if you get stuck with issues:

- **(step 1:) Read the source code:** Everything given to you is a part of an open-source project and is open and accessible. Hence, the majority of issues or confusions could be clarified by just looking into the source code. We expect that you will spend time reading the framework given to you to understand its semantics. Such setup is very common in day-to-day life as a software developer or researcher where you need to quickly read and understand someone else's code. Have you carefully looked at the output and error logs?
- **(step 2) Ask Google:** Try to understand if the issue is related to the coding/Linux/C/CMake/Bash/gcc environment and look for it appropriately. Many of the networking issues that you will have can also be solved with proper Googling, including how to capture packets and debug network issues.
- **(step 3) Ask in the Discussion forum:** if the first two steps do not yield any results, then you can ask in the discussion forum. However, you should not ask for a direct solution to the milestones. Try to ask about specific issues. We or other fellow students will jump in to answer your question. However, be prepared to hear that *"debugging such issues is part of your coding milestones"*. Building any real system is a complex job, and you need to put in the time to solve certain issues. There is no way around it.
- **(step 4) Ask in the lab session:** If the issue is not time critical you can ask in the lab sessions and we can help you out there. Still here, be prepared to hear that *"debugging such issues is part of*

your coding milestones”.

- **(step 5, after exhausting all other possibilities) Reach out to us:** If you believe that you are really stuck and there is something fundamentally wrong (hey, it can happen, no one is perfect ;)) please reach out to us. However, make sure to include what you have tried to do to solve the problem, what possibilities you have investigated, and what you suspect is wrong. Try to include as much information as possible. Show us that you have thought about the problem, and tried really hard to debug it.

What to include when asking for help? Do not just paste the print-screen and say that it does not work, please help. Please show us that you are putting in the effort to solve the problem, not just dumping your problem on us. So, when asking for help please provide details like:

1. What is the problem?
2. When does it happen? Is it non-deterministic or happens every time?
3. What possibilities have you tested to narrow down the error conditions?
4. Have you Google the error code and error message? Did you find something useful?
5. What do you suspect is happening?

Did I mention do not paste print-screens in the Discussion forums when asking for help. Copy paste the log and code.

2.3 Late/Slip days

In the course you have **3 slip days** that you can use to extend the deadline of code/quiz submissions (milestones 1, 2, and 4 *only*). There will be no extensions after that, and you will be awarded zero points. Unfortunately, you **cannot** use these days to move the interview dates. They are fixed, and you need to show whatever you have on that day.

In a group setting, the individual slip days (the left) will be counted and divided by the members of the group and rounded up. So, suppose a 3-member group has {1, 3, 3} slip days left among the members, then the whole group has $(1 + 3 + 3)/3 = 3$ days worth of slip days. If it was {1, 2} slip days (a two-member group) then the group will get $(1 + 2)/2 = 2$ days worth of slip days.

2.4 [Important] Group Management Logistics

The assignments are spread out based on their perceived complexity and the amount of code that needs to be written. Too many times we have seen students showing up a day before the deadline saying that the team member decided not to work, or dropped out of the course - **we cannot help a day before the deadline**. You had a week or more of time to inform us.

Making sure that the group works is your responsibility. *How to do it?*

1. Schedule one or more weekly meetings to work on the project. If one or more members do not show up in one or more meetings without telling why, inform us immediately and notify them that they are no longer part of the group.
2. Meet early and plan ahead who does what and when to synchronize. Use a version control system like git to keep track of issues, and updates.

3. Keep a detailed log (written or digital) of working plans, questions, and doubts that you want to ask - ask them in the lab session, and make sure one or more (preferably all) members of a group attend the lab session regularly.
4. We will not be responsible for resolving disputes in your group. If you are having trouble then contact us early enough with the detailed evidence of what happened.

Extensions: We will consider exceptional circumstances like COVID-19-related interrupts, however, beyond that there is very limited flexibility. Any deadline extensions must be coordinated with your study advisor, and ask them to contact us for an extension. We, as the teachers, can not grant unilateral deadline extensions without asking the study coordinator.

3. Assessment Criteria

Each milestone is worth some points, and in total, you can earn 50 points for the complete project. You can earn additional marks from bonuses, but you are restricted to **2 bonuses max** with a maximum of **additional 20 points** (on top of the maximum of 50 points from the project). Hence, the total points achievable from the project work is 70.

Here is the evaluation plan for the milestones (see individual milestones below in the Handbook for more details):

- **Milestone 1** is evaluated as a graded Canvas quiz.
- **Milestone 2** is evaluated as a graded Canvas quiz and a code upload.
- **Milestone 3** is evaluated with a group interview. All members of the group need to be presented in the interview session to answer questions.
- **Milestone 4** is evaluated as a graded Canvas quiz and a code upload.
- **Milestone 5** is evaluated with a group interview. All members of the group need to be presented in the interview session to answer questions.

3.1 Grading guidelines

"The easiest way to get complete marks is to have a working code" - past successful ANP students.

There are three types of assessments in this course:

1. **A canvas page quiz** that you need to fill out to answer the questions asked. This mode will be used for milestones 1, 2, and 4 (partial). Plus, the code upload.
2. The second form of the assessment takes a more interactive form. We will conduct an **interview for milestones 3 and 5** where you and your group member will be interviewed by us for 10-15 minutes. We will ask questions to test your implementation and your understanding of the assignment material.
3. The third form of assessment is **code evaluation**, where we will look at your source code. Based on the quality and completeness of the code, we will award points.

Canvas quiz consists of objective questions worth 1 or 2 points (says so on the question). There are 4 options, and only 1 is true. Pick the right one. There is no negative marking. Milestones 1 and 2 are graded automatically on Canvas as multi-choice quiz questions. Evaluation criteria for the other two modes are shown below:

	Interview	Code quality
Satisfactory	Majority of questions answered, needing hints in answering, with limited understanding of broader concepts.	The basic working code adheres to the specification given. Many event ordering assumptions were made. Primitive error handling. Limited attention to the code quality (long functions, random variable names, unspecified numbers used, zero/limited documentation in the code). Unreasonable constructs used (e.g., busy looping for some condition to be true).
Good	All questions answered, needing hints in answering, with limited understanding of broader concepts.	The working code adheres to the specification given. A few event ordering assumptions were made. Moderate error handling. Attention to the code quality (small functions, no random variable names or unspecified numbers used, limited documentation in the code). No unreasonable constructs were used (e.g., busy looping for some condition to be true).
Very good	All questions answered, showed some/limited broader understanding of concepts.	The working code adheres to the specification given. No event-ordering assumptions or unreasonable constructs were used. Good error handling. Good attention to code quality to make it legible.
Excellent	All questions were answered, a good understanding of broader concepts.	The working code adheres to the specification given. No event-ordering assumptions or unreasonable constructs were used. Very good error handling, thinking about many underspecified error conditions. Good attention to code quality to make it legible, mostly following the kernel coding guidelines.

Broad understanding here means, does the group understand what happens outside their framework in the networking domain (Linux stack), concepts from the class (can they relate to that), etc. **Hints** here mean that the group needed specific hints to answer the question, or do not remember what they have implemented, or other details that they should have known from the implementation of the code.

Protocol for interview assessment:

- We will first ask if your group has a working project
 - **If yes**, show us a demo and explain what is happening
 - **If not**, then demonstrate to us to the best of your abilities (explain, code) where you are in the milestone.

- Explain various pieces of the code that you have written and what design and logic you have considered implementing and why (part of the broad understanding).
- The code will be judged on the generality (if you are making unreasonable assumptions, like event ordering), error handling, hardcoding numbers, unexpected sleep/wait conditions to avoid race conditions, etc. We can not give a comprehensive list of criteria on which your code will be evaluated, however, we are reasonable people and we will explain what is being evaluated and why.
- Follow a coherent coding style, something like <https://www.kernel.org/doc/html/latest/process/coding-style.html>. A very poorly written code will incur a penalty.
- You are graded on (1) if the code is working; *and* (2) how well do you understand what is implemented, what are the corner conditions, where the code will break, how it could have been improved, and general knowledge - both team members, even if you split the work; (3) how well do you answer questions during the interview.

More interview logistics will be published as we approach the interview week.

Note on assessment and marking: Students are often under the impression that since they wrote a lot of code, and spent too much time in front of the computer they should get marks. Unfortunately, this is not how programming assignments work. **You do not get marks just because you wrote code.** If you spend many hours writing code that does not work and does not adhere to what is asked in the assignment, you will get zero. You get marks for writing code that adheres to the given functional specification. The project that is given to you has been done by us carefully and we know how much effort it should take as a team. This is a 6 ECTS course, and you need to put time and effort to get it done. A working code is the best proof of your time, energy, and effort put into this course. Randomly modifying the framework, and hoping that the code works will get you nowhere.

3.2 Specific notes on partial assessment for Milestones

It might be the case that you do not have a working code for milestones 3-5 at the time of the interview. So here is the protocol for assessment

1. You will first be interviewed on your general understanding of the milestone and what you tried to implement
2. If you demonstrated sufficient expertise then you can demonstrate your partial working code
3. In case you did not demonstrate sufficient understanding of the milestone, we will not review the code, and the maximum points awarded for this milestone would be whatever you achieve in the interview.

Milestone 5: in case you fail to get your code working to run milestones 3-4 (TCP stack), an alternate plan is to perform the experiments for M5 with Linux network stack. All evaluation criteria will remain the same. **However, the maximum achievable marks will be restricted to 50% of the maximum marks possible for the milestone.**

4. Background Network Reading Material

This course is built on top of foundational networking knowledge which we assume that you have acquired in other courses. However, if this is not the case then the following are the online and offline

resources from where you can acquire the necessary knowledge. Knowing these concepts will not only help you in the practical part but also in the lectures to follow the details more easily:

- Networking basics: Protocols, Layer models, TCP/IP basic, Ethernet, TCP, IP, ARP, ICMP, packet formats, the concept of encapsulation.
 - See: Computer Networks (5th Edition) by Andrew Tanenbaum (Author), David Wetherall (Author) sections: 4.3 (Ethernet), 5.6 (IP), 5.6.4 (ICMP and ARP), 5.6.2 (IP addresses, NAT), 6.1.3 (Berkeley sockets), 6.5 (TCP)
- Operating system basics: see notes from Operating Systems (X_405067), what is a process, shared library, address space
 - See: Operating Systems: Three Easy Pieces, chapters 4 (process), 13 (Address Space), 15 (Address translation), <https://pages.cs.wisc.edu/~remzi/OSTEP/>
- Programming and threading basics: Good working knowledge of C/C++, socket background
 - **[socket]** Beej's Guide to Network Programming Using Internet Sockets, <https://beej.us/guide/bgnet/> (html book: <https://beej.us/guide/bgnet/html/>) Please read until the chapter Client-Server Background)
 - **[thread and locking]** Operating Systems: Three Easy Pieces with code examples, chapter 27 (Thread API) and 30 (Condition Variables), <https://pages.cs.wisc.edu/~remzi/OSTEP/>. Please browse other sections in the concurrency section to help build a better understanding.
- Hardware basics: see notes from Computer Organization (XB_40009) : CPU, devices, interrupts, memory architecture, caches
 - <https://github.com/animeshtrivedi/animeshtrivedi.github.io/raw/master/files/2019/2019-comparch-io.pdf>

5. Project ANP Netstack: Building my own networking stack!

You are provided a simple TCP server-client application (server-client folder) where the client connects to the server and sends a buffer to the server. The server sends the same buffer content back to the client. The client and server both check for a predefined pattern on the buffer. In the case when using the Linux in-kernel netstack implementation this check matches. ***The goal of your netstack implementation is to run the same program without modification on your networking stack.***

To make this happen, we are going to build the ANP netstack as a **shared library** and use the **Linux TUN/TAP infrastructure** [1, 2, 3, 4, 5]. The TUN/TAP interface is used to read/write packets directly from the userspace. It emulates a simple point-to-point ethernet device that can be used to read/write packets to and from the user application. For the rest, we can configure the TUN/TAP device using the IP forwarding rules to ensure that the packets are routed properly to the outside world. Please see the documentation (references 1-5) to get a better understanding of TUN/TAP devices. TUN devices are for IP packets (L3, layer 3, in the OSI reference model), and TAP devices are for raw Ethernet packets (L2). We will be using the TAP device in this course. A deep understanding of the TUN/TAP devices is not needed to complete the project work.

The ANP netstack shared library will be loaded before the client program starts using the `$LD_PRELOAD` bash environment variable. This environment variable lets our framework take over the networking system calls before they go to the kernel. Hence, in this case, we will implement these networking systems calls in the shared library, without letting them go to the Linux kernel. **See figures 2 and 3 for an overview of the overall setup.**

Important: *Whenever in doubt, read the source code. We cannot emphasize this part enough. The whole source code from the framework is your assignment.* There are comments, and links to further helpful resources in the source code. Many errors that you will encounter are typical networking errors, a simple googling will give you sufficient hints on how to debug the situation. Use any and every debugging tool available on Linux. If you do not know which tool to use, please ask us. We are happy to provide hints.

[1] Universal TUN/TAP device driver, <https://www.kernel.org/doc/Documentation/networking/tuntap.txt>

[2] Tun/Tap interface tutorial, <https://backreference.org/2010/03/26/tuntap-interface-tutorial/>

[3] TUN/TAP devices on Linux, <http://recolog.blogspot.com/2016/06/tuntap-devices-on-linux.html>

[4] TUN/TAP Interface, <https://hechao.li/2018/05/21/Tun-Tap-Interface/>

[5] Understanding TUN TAP Interfaces, <http://www.naturalborncoder.com/virtualization/2014/10/17/understanding-tun-tap-interfaces/>

4.1 Milestone 1: *Welcome to the machine*

Type: Individual

Maximum points: 10

Where and what: Take the canvas quiz

Deadline: see the course canvas page

The first milestone is about getting the coding infrastructure up and working. You should be able to run the demo arping example and answer basic questions about how the arp protocol works.

Setup: The primary development environment for the project is Linux. The easiest way to get a Linux system is to install one on your laptop/computer, but a virtual machine (like virtual box) should do as well. At the start, we are deploying code in a server-client setting. This server and client can run on the same machine, or two different physical machines, one physical machine, and one virtual machine, and various other possible combinations. Unfortunately, this complicates the setup a bit, we will try to provide some helpful guidelines here.

Terminology:

- **Host** - the host machine where you run the virtual box program. A host machine could be running Linux, Mac, or Windows programs.
 - Commands that run here are shown as **user@host:~\$**
- **Virtual machine** - the virtual machine. Inside the virtual machine we run Ubuntu 18.
 - Commands that run here are shown as **vuser@vm:~\$**

Simplest way to get started is to have a Linux host and have a Linux virtual machine (VM) for development, which can be achieved using virtual box (or QEMU). You are required to have a Ubuntu 18.04 LTS setup (this is where we have done most of the testing. In theory, the program should work anywhere, but we cannot test all Linux distributions). In a VM like Virtual Box (<https://www.virtualbox.org/>). You can see how to install a Ubuntu VM here :

https://linuxhint.com/install_ubuntu_18-04_virtualbox/

The ISO can be downloaded from here: <https://releases.ubuntu.com/18.04/>

For M1/M2 macbooks: We have not tested M1/M2 macbooks as we do not have access to it. If you have one, and setting up Virtual Box on it (<https://www.virtualbox.org/wiki/Downloads> , see the *Developer preview for macOS / Arm64 (M1/M2)* file download) then please notify us, and we will help to ensure that either you have a working setup on your mac/or other machine we can arrange.

Important note on the disk and memory size: Use at least **30GB** of virtual disk space for the installation (it does not take immediately 30GB of disk space, but will allocate gradually as needed), and **2-4 GB** of DRAM. The more you can afford the better. In case you run out of memory errors (Out-of-memory, OOM killed in your dmesg command log), either (i) increase the memory size more; and (ii) enable a large swap space (2-4GB) <https://linuxize.com/post/how-to-add-swap-space-on-ubuntu-18-04/>. Running out

of disk or memory resources will show up in a subtle way where unexpected commands will fail without any useful warnings. In case you run out of disk space, you can resize the disk image as shown here <http://www.jochenhebbrecht.be/site/2016-07-11/windows/resizing-disk-drive-in-ubuntu-guest-system-running-a-virtual-box-windows-host-syt>

Setting up the networking for the virtual machine: There are multiple ways to set it up. Also which mode to choose depends on the host system you have. There are three modes (1) Bridged mode - where the networking card of the virtual machine shows as another machine on the network and interacts with the rest of the network (and outside to the internet directly); (2) NAT mode, where the host machine does network NAT translation between the outside world and the virtual machine; and (3) host-only adapter, which is a special case of NATing, where the virtual machine has a special connection to the host special adapter.

We will be using the third mode with a virtual box. Here is the step by step guide:

Step 1: Creating a host-only adapter: We first create a host only adapter in the virtual box. Click on the “Tools”, and see if there is already an adapter available such as “vboxnet0”. If not, then create one by clicking on the “Create tab”.

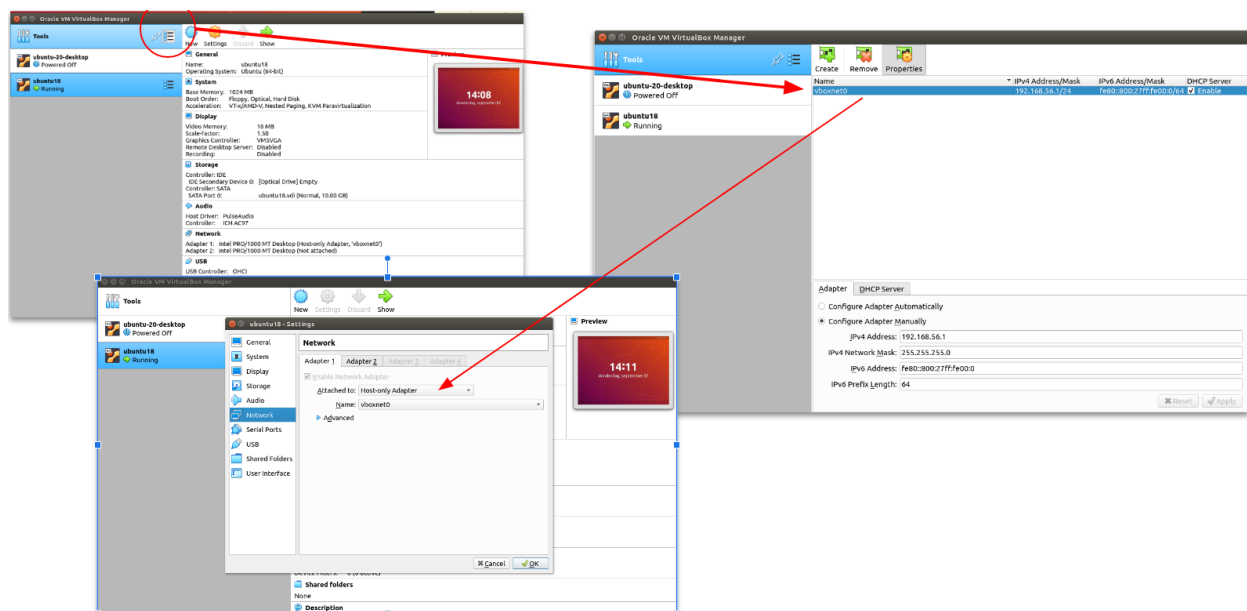


Figure 1: Virtual Box setting with a private network with the host. (Apologies for the low resolution)

If yes (a card already exists), then make sure that the DHCP server box is checked. You should select Configure Adapter Manually and put 192.168.56.1 as the IPv4 address, and 255.255.255.0 as the IPv4 mask. On the DHCP server tab, you should put the following values (most likely there are already there):
Server address: 192.168.56.100
Server mask: 255.255.255.0
Lower bound address: 192.168.56.101

Upper bound address: 192.168.56.254

Step 2: Attaching to the virtual machine: Setup a host-only adapter in your virtual machine settings, see the setup below:

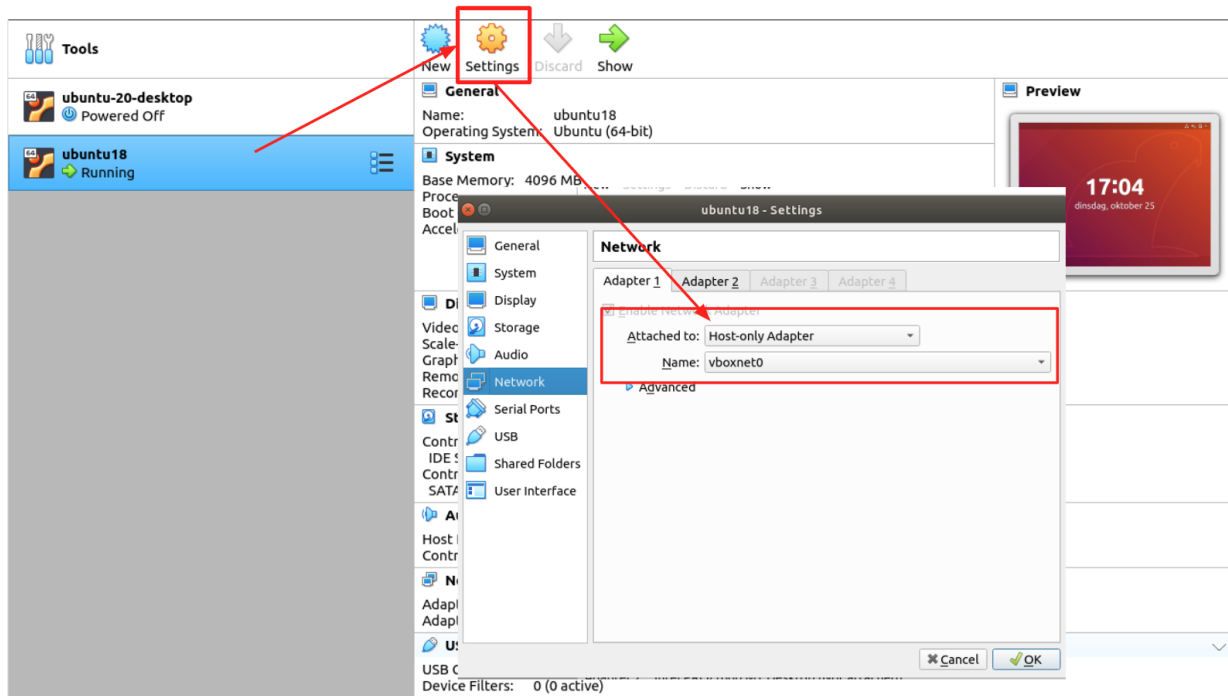


Figure 2: Attaching a host-only adapter to your virtual machine setting (before booting).

Step 3: Enabling ssh access: With this setup, your Ubuntu-18 VM is connected to your host machine with a bridge NIC (*vmbox0*) but your VM will not have Internet connectivity. We can enable that by setting up IP forwarding on your Linux host between *vmbox0* and your host's internet link (in my case, *wlp2s0*).

Log in your virtual machine using ssh command like this (this is to avoid “too many authentication failures with many ssh keys” as SSH uses key based authentication first):

```
user@host:~$ ssh -o PreferredAuthentications=password -o PubkeyAuthentication=no 192.168.56.101
```

When copy pasting bash commands - pay big attention to “.” (dots”), spaces between keywords, and the next line character. Bash commands are always single line. Don't line break them.

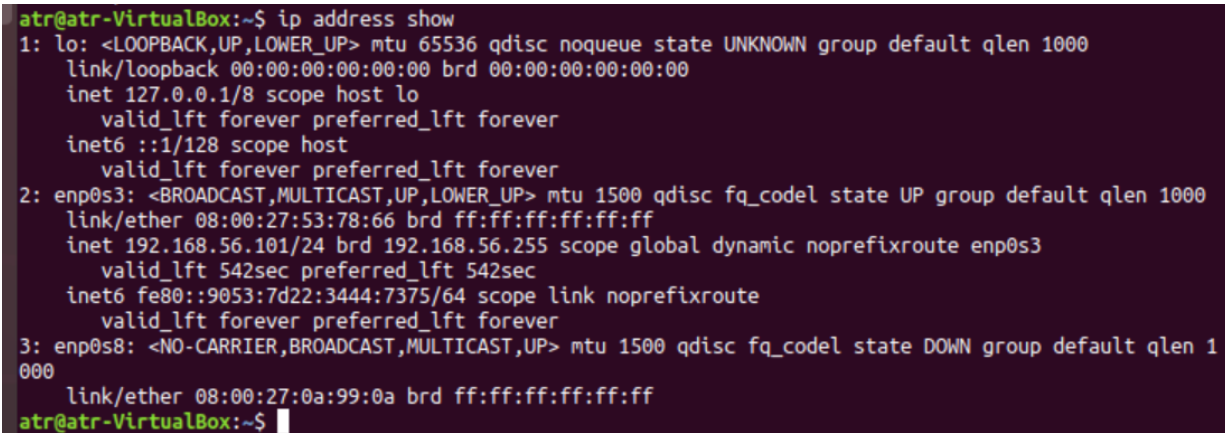
You can set up the public key infrastructure separately to do passwordless access to the machine. See <https://kb.iu.edu/d/aews> (or ask us in the lab).

Most likely your virtual machine has this IP (192.168.56.101) which was the same IP starting range that we used with the DHCP server setting (lower bound address).

In case this is not so, then we need to find out the IP address by logging into the virtual machine. Please use the GUI of the machine on the virtual box, login using the username and password credentials that you set up during the installation. Inside the virtual machine, execute the following command:

```
vuser@vm:~$ ip address show
```

As you can see in the example below in my virtual machine, I have IP of 192.168.56.101 set for enp0s3 host-only NIC adapter inside the VM



```
atr@atr-VirtualBox:~$ ip address show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:53:78:66 brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.101/24 brd 192.168.56.255 scope global dynamic noprefixroute enp0s3
        valid_lft 542sec preferred_lft 542sec
    inet6 fe80::9053:7d22:3444:7375/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
3: enp0s8: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc fq_codel state DOWN group default qlen 1000
    link/ether 08:00:27:0a:99:0a brd ff:ff:ff:ff:ff:ff
atr@atr-VirtualBox:~$
```

Figure 3: Output of the ip address show command that shows the IP address

You can use this IP address to log into the virtual machine as shown previously from the host. If you get some other IP address, then please use that IP in all the following command sequences.

Step 4: Enabling Internet access from inside the VM:

We need to enable IP forwarding on the host so that the host machine sends out the packets from the virtual machine to the outside world (NATing). Here is an example of a command sequence. “w1p2s0” is the NIC on the host which is connected to the outside world.

```
# a comment starts like this

# enable IPv4 forwarding on the host machine
user@host:~$ cat /proc/sys/net/ipv4/ip_forward
0
user@host:~$ sudo su
user@host:# echo 1 > /proc/sys/net/ipv4/ip_forward
root@atr-xps-13:/home/atr# exit
user@host:~$ cat /proc/sys/net/ipv4/ip_forward
1

# 192.158.56.0/24 (is the range of IPs that are managed by the DHCP server)
```



```

user@host:~$ sudo iptables -I INPUT --source 192.168.56.0/24 -j ACCEPT
user@host:~$ sudo iptables -t nat -I POSTROUTING --out-interface wlp2s0 -j MASQUERADE
user@host:~$ sudo iptables -I FORWARD --in-interface wlp2s0 --out-interface vboxnet0 -j ACCEPT
user@host:~$ sudo iptables -I FORWARD --in-interface vboxnet0 --out-interface wlp2s0 -j ACCEPT

```

After this you also need to tell your VM how to send a packet out. In the example below, when you ping an IPv4 address of 1.1.1.1, the VM does not know where to send this packet, which NIC to use, which MAC address to use, etc. We can verify this lack of knowledge by printing the kernel routing tables, which only know how to handle packets for 169.254.0.0/16 and 192.168.56.0/24 ranges by putting them out to enp0s3 NIC. It does not know how to send a packet to Google, for example, which might have an IP of 172.217.168.206. To resolve this, we can then add a default gateway path for “any” packet (0.0.0.0/) to be put out to that NIC as well. Recall that IP CIDR matching is done as precisely as possible. Hence, routes that match the most will be used. Once, we put this entry, the system is able to ping 1.1.1.1 IP (which is actually Cloudflare's public DNS IP address. *Do you want to find out who owns 8.8.8.8?*)

```

# can the VM ping 1.1.1.1, the Cloudflare's DNS address?
vuser@vm:~$ ping 1.1.1.1
connect: Network is unreachable

# see if the kernel routing entries are set?
vuser@vm:~$ route -n
Kernel IP routing table

```

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
169.254.0.0	0.0.0.0	255.255.0.0	U	1000	0	0	enp0s3
192.168.56.0	0.0.0.0	255.255.255.0	U	100	0	0	enp0s3

```

# add the entry to route "all" packets to enp0s3
vuser@vm:~$ sudo route add default gw 192.168.56.1 enp0s3

# check again, this time you should see a new entry
vuser@vm:~$ route -n
Kernel IP routing table

```

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
0.0.0.0	192.168.56.1	0.0.0.0	UG	0	0	0	enp0s3
169.254.0.0	0.0.0.0	255.255.0.0	U	1000	0	0	enp0s3
192.168.56.0	0.0.0.0	255.255.255.0	U	100	0	0	enp0s3

```

# ping should work now!
vuser@vm:~$ ping 1.1.1.1
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
64 bytes from 1.1.1.1: icmp_seq=1 ttl=56 time=4.10 ms
64 bytes from 1.1.1.1: icmp_seq=2 ttl=56 time=4.60 ms
^C

```

```

--- 1.1.1.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 4.103/4.354/4.606/0.260 ms

```

At this stage: (1) the host can talk to the VM (ssh works); (2) VM can talk to the internet - ping works. You are ready to deploy the given framework.

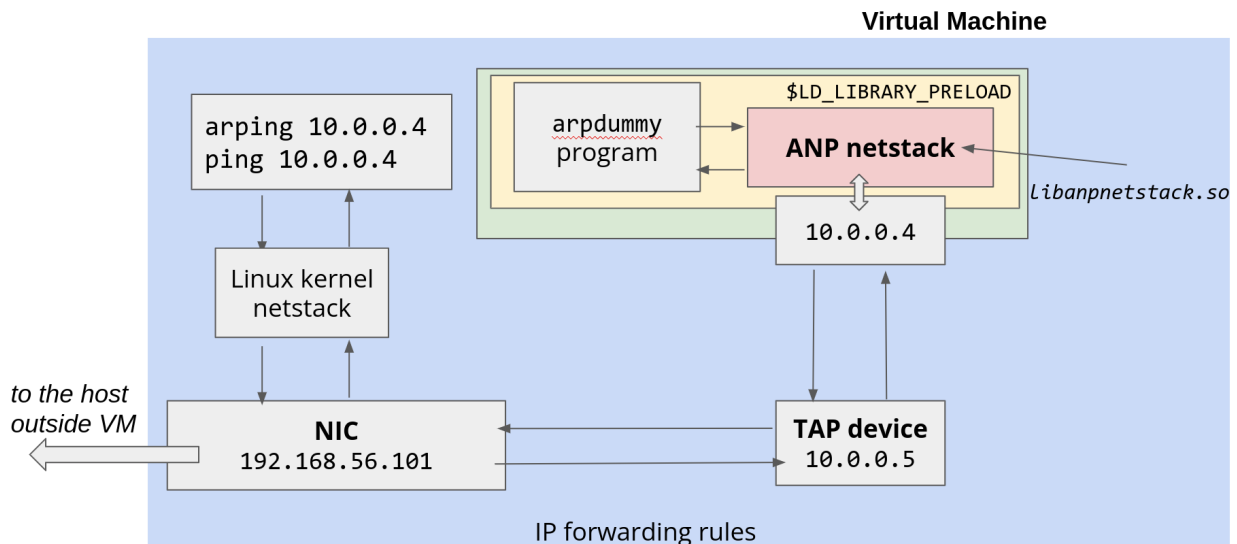


Figure 4: the networking setup used in this project.

Windows Specific Instructions:

On windows, you can create a virtual machine using Virtual Box as usual.

- Install VirtualBox
- Download the **Ubuntu 18.04 LTS** iso (<https://releases.ubuntu.com/18.04/>)
- In VirtualBox: Create a **new virtual machine** with a name of your choice and select the downloaded iso, then press "Next"
- Choose a **username and password** (You can leave everything else unchanged). Then press "Next".
- Choose between **2048 MB** and **4096 MB** of **memory**, and the number of **processors** to use (around half of the resources you have on your machine should be enough). Then press "Next"
- Select at least **30 GB of Disk Size**. Then press "Next".
- You can now create your virtual machine.

With these settings, your virtual machine should work out of the box and be able to connect to the internet. You can then download the code skeleton and follow the rest of the setup.

Build environment and initial code skeleton: ANP networking code is developed as a shared library using the CMake build environment. The initial code that is given to you contains some basic infrastructure code (timer, system user buffer (subuff - socket user buffer, similar in spirit to SKB in the linux kernel), with ARP protocol implementation. For the first week's assignment your goal is to set up,

and successfully run the ARP example, set up your coding environment, read the code and understand, and then eventually answer quiz questions on Canvas for week 1.

If you did a fresh installation of Linux, then you may want to install the necessary packages (please google if you run into unexpected errors, missing files, or tools).

```
# install core development packages
vuser@vm:~$ sudo apt-get install git build-essential cmake libcap-dev arping net-tools
tcpdump
```

Setup and compile the code:

```
# get into the source code directory
vuser@vm:~$ cd anp-netskeleton

# do a cmake build
vuser@vm:~/anp-netskeleton$ cmake .
-- The C compiler identification is GNU 7.5.0
-- The CXX compiler identification is GNU 7.5.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /home/atr/anp_netskeleton

# make it
vuser@vm:~/anp-netskeleton$ make -j
Scanning dependencies of target anp_server
Scanning dependencies of target anpnetstack
Scanning dependencies of target anp_client
[ 10%] Building C object CMakeFiles/anp_server.dir/server-client/common.c.o
[ 10%] Building C object CMakeFiles/anp_server.dir/server-client/tcp_server.c.o
[ 15%] Building C object CMakeFiles/anp_client.dir/server-client/tcp_client.c.o
[ 21%] Building C object CMakeFiles/anp_client.dir/server-client/common.c.o
[ 26%] Building C object CMakeFiles/anpnetstack.dir/src/init.c.o
```

```

[ 31%] Building C object CMakeFiles/anpnetstack.dir/src/utilities.c.o
[ 36%] Building C object CMakeFiles/anpnetstack.dir/src/anpwrapper.c.o
[ 42%] Building C object CMakeFiles/anpnetstack.dir/src/ip_rx.c.o
[ 47%] Building C object CMakeFiles/anpnetstack.dir/src/arp.c.o
[ 52%] Building C object CMakeFiles/anpnetstack.dir/src/tap_netdev.c.o
[ 57%] Building C object CMakeFiles/anpnetstack.dir/src/timer.c.o
[ 63%] Building C object CMakeFiles/anpnetstack.dir/src/anp_netdev.c.o
[ 68%] Building C object CMakeFiles/anpnetstack.dir/src/route.c.o
[ 73%] Building C object CMakeFiles/anpnetstack.dir/src/icmp.c.o
[ 78%] Building C object CMakeFiles/anpnetstack.dir/src/subuff.c.o
[ 84%] Building C object CMakeFiles/anpnetstack.dir/src/ip_tx.c.o
[ 89%] Linking C executable build/anp_client
[ 94%] Linking C executable build/anp_server
[ 94%] Built target anp_server
[ 94%] Built target anp_client
[100%] Linking C shared library lib/libanpnetstack.so
[100%] Built target anpnetstack

```

```

vuser@vm:~/anp-netskeleton$ ls ./lib/
libanpnetstack.so  libanpnetstack.so.1  libanpnetstack.so.1.0.1
# success, that is it!
vuser@vm:~/cd anp-netskeleton$

```

```

# In case you run into some compilation issues, you should always clean the build directory
vuser@vm:~/anp-netskeleton$ make clean
vuser@vm:~/anp-netskeleton$ rm -rf Makefile  Testing/  anpnetstack.cbp  build/
cmake-build-debug/ CMakeCache.txt

```

Once done compiling the code, then (see the README.md and check the bash shell scripts in the ./bin/ folder). You need to do three things:

1. Make a TAP/TUN device
2. Disable IPv6
3. Setup packet forwarding rules

```

# check the bin directory
vuser@vm:~/anp-netskeleton$ ls ./bin/
sh-debug-anp.sh  sh-disable-ipv6.sh  sh-hack-anp.sh  sh-make-tun-dev.sh
sh-setup-arpserver.sh  sh-setup-fwd.sh

# Step 1: setup the TUN/TAP device
vuser@vm:~/anp-netskeleton$ ./bin/sh-make-tun-dev.sh
+ sudo mknod /dev/net/tap c 10 200
[sudo] password for atr:
+ sudo chmod 0666 /dev/net/tap

# Step 2: Disable IPv6
vuser@vm:~/anp-netskeleton$ ./bin/sh-disable-ipv6.sh
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1

# Step 3: Setup NATing between the TUN/TAP device and virtual machine - use the NIC which
connects to the host
vuser@vm:~/anp-netskeleton$ ./bin/sh-setup-fwd.sh enp0s3

```



```

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default
qlen 1000
    link/ether 08:00:27:53:78:66 brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.101/24 brd 192.168.56.255 scope global dynamic noprefixroute enp0s3
        valid_lft 501sec preferred_lft 501sec
3: enp0s8: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc fq_codel state DOWN group
default qlen 1000
    link/ether 08:00:27:0a:99:0a brd ff:ff:ff:ff:ff:ff

```

You can do the same on the host machine to get information about the host-only bridge NIC. For example, on my host, the vboxnet0 is connected to the virtual machine:

```

# shows all valid link/NIC IP and MAC addresses (virtual as well as physical)
user@host:~$ ip address show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: wlp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
qlen 1000
    link/ether 9c:b6:d0:97:92:47 brd ff:ff:ff:ff:ff:ff
    inet 145.108.76.180/22 brd 145.108.79.255 scope global dynamic noprefixroute wlp2s0
        valid_lft 1436sec preferred_lft 1436sec
    inet6 fe80::ed60:646:91ef:7daf/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
69: enx8c04ba560511: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc fq_codel state DOWN
group default qlen 1000
    link/ether 8c:04:ba:56:05:11 brd ff:ff:ff:ff:ff:ff
70: vboxnet0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group
default qlen 1000
    link/ether 0a:00:27:00:00:00 brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.1/24 brd 192.168.56.255 scope global vboxnet0
        valid_lft forever preferred_lft forever
    inet6 fe80::800:27ff:fe00:0/64 scope link
        valid_lft forever preferred_lft forever

```

So now if I do an ARP ping for the vboxnet0 IP address inside the virtual machine:

```

# arp ping to the host bridge (using the IP address from the host what is highlighted)
vuser@vm:~$ sudo arping 192.168.56.1 -c 2
ARPING 192.168.56.1
60 bytes from 0a:00:27:00:00:00 (192.168.56.1): index=0 time=484.786 usec
60 bytes from 0a:00:27:00:00:00 (192.168.56.1): index=1 time=770.512 usec

--- 192.168.56.1 statistics ---
2 packets transmitted, 2 packets received, 0% unanswered (0 extra)
rtt min/avg/max/std-dev = 0.485/0.628/0.771/0.143 ms

```

As you can see that the arping command (that implements the ARP protocol) has resolved an IP address to its 48-bits MAC address correctly. Naturally all this traffic is going through the Linux kernel netstack. How do we get traffic to our libanp library code? We ping the IP address of the TUN/TAP device. We have created a device in the kernel, but we have not set it up or made it active. So, let's start with that. Since we cannot execute a shared library, we need to attach the library to a process. So we created a dummy process (binary file) called arpdummy. You can open the source code of arpdummy.c file and it just sleeps indefinitely unless canceled by calling ctrl + c.

```
# make a binary that links to the shared library
vuser@vm:~/anp_netskeleton/bin$ ./sh-setup-arpserver.sh
++ gcc ./arpdummy.c -g -o arpdummy
++ set +x
-----
The ANP netstack is ready, now try this:
1. [terminal 1] ./sh-hack-anp.sh ./arpdummy
2. [terminal 2] try running arping -I tap0 10.0.0.4
```

Terminal 1:

```
# run the dummy arp process
vuser@vm:~/anp_netskeleton/bin$ sudo ./sh-hack-anp.sh ./arpdummy
... ???
```

Terminal 2:

```
# arping the private IP address of the application
vuser@vm:~$ arping -I tap0 10.0.0.4
...???
```

You will see some output (marked as "...???") on both terminals at this point. Once you have the code working, please browse around in the source code, make yourself familiar. Make sure you understand what is happening with the two terminals. Do you understand where the IP address 10.0.0.4 comes from?

Then attempt the milestone 1 canvas quiz (while having your code running on the side). We will ask you to modify the code and ask about the output. The current implementation of ARP is also used when first establishing the TCP connection. So it is important that you spend time understanding how ARP works.

Read about network debugging tools: tcpdump, arping, ping, netstate. Most important among them is tcpdump. Here is an example of such a tutorial: <https://danielmiessler.com/study/tcpdump/>. Use of tcpdump will come in handy when you want to debug your TCP/IP/ICMP packets, and want to print headers. Tcpdump and wireshark do the same thing. These tools work on the underlying same principle of packet capturing in Linux.

Make sure you understand the Socket user buffer semantics: Spend some time to understand how a packet is constructed and transmitted. Specifically what is the semantics of various functions to

manipulate the struct subbuff (src/subbuff.h) which mimics the more daunting and infamous the Linux kernel SKB structure,

<https://web.archive.org/web/20210609180906/https://vger.kernel.org/~davem/skb.html>

There is a **anp_sub_note_clarification.pdf** file in the Files section. Please study it carefully to understand how packet encapsulation and header construction is done for IP and Ethernet packets. All packets: ARP (right now), later ICMP and TCP, IP and ETH headers and payloads are put in a single sub structure (can be allocated for a max packet size of 1500 bytes). You need to be careful where you put your protocol header and payload.

In the first milestone you are supposed to build and install packages. This is a relatively simple milestone, and we would **strongly recommend** making use of this time to familiarize yourself with the concepts and source code.

Milestone summary: What needs to be done:

1. Setup the given infrastructure on a virtual machine - make sure the Internet works inside your VM. Install necessary packages and coding environments for yourself.
2. Compile and deploy the given framework.
3. Get the arping command work.
4. Study the framework code, and arp implementation to understand how a packet format is written, and how a packet is sent and received. Follow and debug the code to ensure that you understand it. Progressive milestones will get harder, and really depends on the understanding of these concepts. Also, refresh your networking concepts. Do you understand how the "subbuff" structure works? What does __attribute__((packed)) do?
5. Have a working code next to you when you attempt the Canvas quiz. **The quiz may ask you to modify and run the code under specific conditions and ask what happens.**

4.2 Milestone 2 : *Is anyone out there?*

Type: individual

Maximum points: 10

Where and what: (1) Take the canvas quiz and (2) code upload on canvas (zip file)

Deadline: see the course canvas page

The second milestone is about implementing the ICMP protocol, which is used in the ping command. ICMP stands for Internet Control Message Protocol and it is used as a diagnostic protocol over IP networks. You will be implementing the ping response message.

The basic source code already has headers and empty source code for ICMP implementation in icmp.[ch] files. Study them carefully and complete the implementation. Pay careful attention to network and host byte ordering. The network packets are always in the network byte order. See this link for more details: <https://linux-kernel-labs.github.io/refs/heads/master/labs/networking.html#conversions>

We are going to use the same setup as Milestone 1 with two terminals:

Terminal 1:

```
# run the dummy arp process, it will start and then crash as soon as it gets the ping packet
vuser@vm:~/anp_netskeleton/bin$ sudo ./sh-hack-anp.sh ./arpdummy
...

DEBUG_IP: (ip_rx, 62) [M2] incoming ICMP packet, further ICMP packet processing is not yet
implemented
arpdummy: /home/atr/anp_netskeleton/src/icmp.c:30: icmp_rx: Assertion `false' failed.
./sh-hack-anp.sh: line 9: 6369 Aborted (core dumped)
LD_PRELOAD="/usr/local/lib/libanpnetstack.so" "$prog" "$@"
```

Terminal 2:

```
# ping the private IP address of the application
vuser@vm:~$ ping -I tap0 10.0.0.4 -c 1
```

After a successful implementation, the above command in terminal 2 should get constant ping responses back like:

Terminal 2 (a successful run example):

```
# ping the private IP address of the application
vuser@vm:~$ ping -I tap0 10.0.0.4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=55 time=8.22 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=55 time=3.98 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=55 time=5.54 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=55 time=5.78 ms
64 bytes from 10.0.0.4: icmp_seq=5 ttl=55 time=4.92 ms
^C
--- 10.0.0.4 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4051ms
rtt min/avg/max/mdev = 3.983/5.694/8.229/1.412 ms
```

Some coding tips, common mistakes, and hints: make sure you understand the return values of many provided functions. In general, network packet processing is a highly asynchronous task. That means, a lot of processing is triggered via events, and you need to try again or wait for certain events or conditions to be true.

For example, pay attention to the `ip_output()` function. When a packet is sent out for the first time for an unknown IP address, the routing implementation internally needs to figure out which MAC address to use. Hence, it will send out an ARP request (you should be able to see this on the wire, and incoming response). Make sure that this is a right request with a right IP address. Otherwise the routing subsystem can never find out the right destination MAC address. What you should see with ARP is : looking for the mac address of the TAP device. The TAP device is the gateway device for all the traffic from your application-private NIC. The incoming ARP response should contain the MAC address of the TAP device. You can verify that by matching that with the `ifconfig tap0`. Often it happens in many systems implementations that the system returns `-EAGAIN`. That means you should try again after "some" time, not immediately in a loop. So I would recommend that you wait for a certain time period (reasonable

time like ~10-100s of milliseconds), and then try again. ***So, did your ip_output() function call returned 0 or -EAGAIN?***

Always install the library: Do not forget to “**sudo make install**” whenever you write a new code. The shared library is installed at /usr/local/lib/ path and picked up from there for linking and loading. Whenever you modify the source code, you need to make it, and install it by calling: **sudo make install**.

How to debug: multiple things can go wrong before a packet is sent out or received successfully. When debugging you can follow a simple protocol like:

1. Is the problem on the sending or receiving side?
2. Check and print if all IP and MAC values are correct?
3. Check if the sub buffer pointers are set properly for the ip_output() function to construct its IP packet?
4. Is the packet transmitted successfully?
5. Do you see the packet on the wire, captured either using tcpdump or wireshark? Do the header values in the captured packet look the same as what you set in the program?

Milestone summary: What needs to be done:

1. Implement the icmp_rx and icmp_reply functions in the source code, with any other utility functions needed to successfully run the ping command
2. Understand how ping command works with your stack
3. Answer the Canvas quiz questions for milestone 2 (you only have one attempt), have the running code next to you. ***The quiz may ask you to modify and run the code under specific conditions and ask what happens.***
4. Zip and the source code with the complete framework, and upload on Canvas within the deadline.
5. **Make sure you include everything with a running code. If we cannot compile your code, you get a zero. You will not get an extension to fix your uploading mistakes.**

4.3 Milestone 3: *Hey you*

Type: group work (2-3 students)

Maximum points: 10

Where and what: (1) Code upload on canvas (zip file) and (2) show up for the in-class interview

Deadline: see the course canvas page

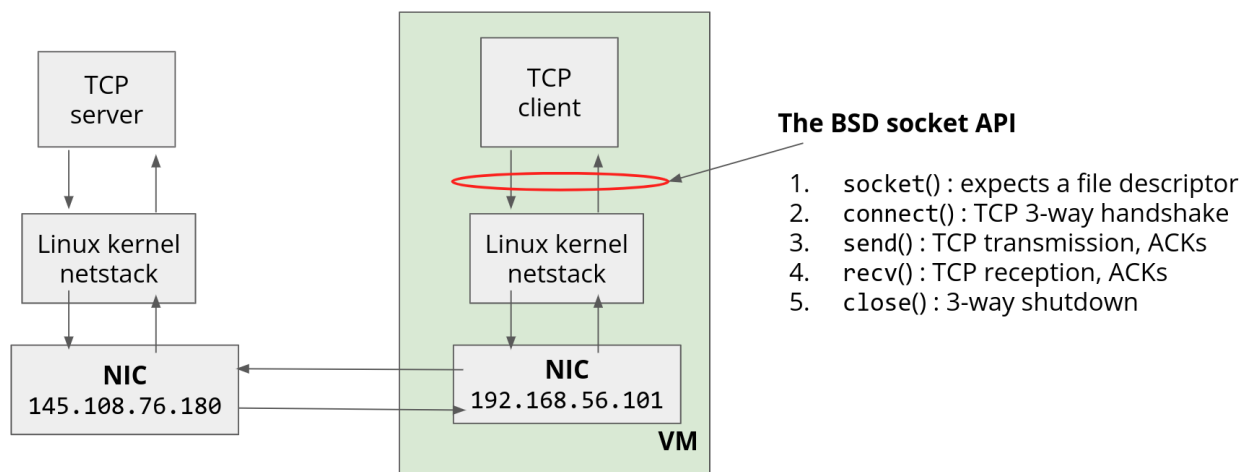


Figure 5: The overall setup and various locations where the TCP server and client program will be running and how they communicate. This is a logical setup, and can be “realized” in multiple ways such as **running server-client on the same VM machine or on two different machines.**

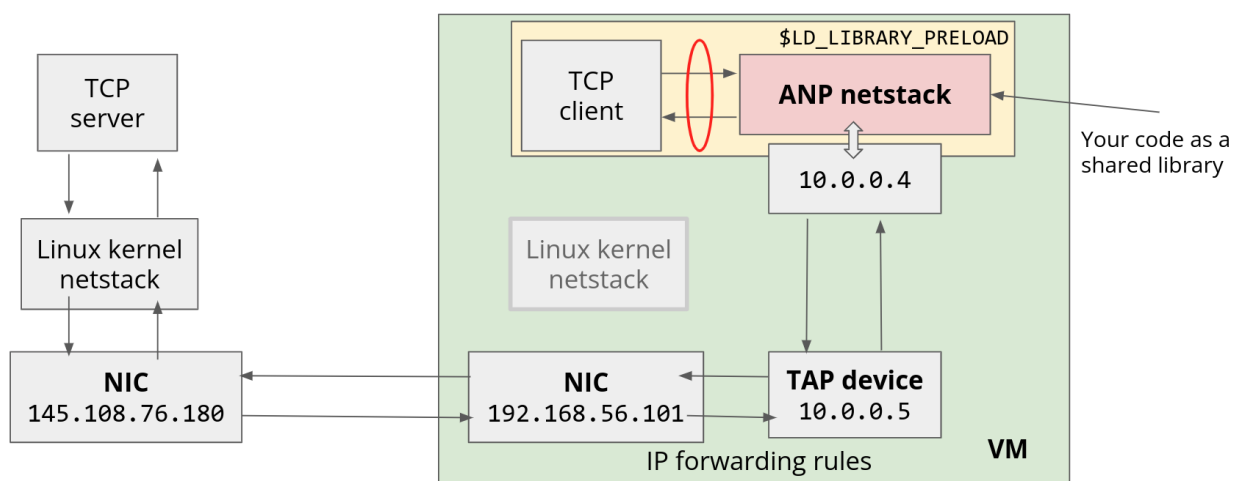


Figure 6: The overall setup and various locations where the TCP server and client program together with the TAP device and ANP networking stack.

You are given a basic TCP server client program also (see the server-client folder in the given framework). Browse through the source code. In a nutshell, a client connects to a server, sends a test buffer written with a predetermined pattern. The server upon receiving the buffer checks the patterns, and echoes the buffer back to the client, where the client checks it again. **You will not modify this program at any point in time. The ANP netstack code only implements the client side calls, specifically - socket, connect, send, recv, and close. You should be able to transparently take over networking calls from this program.**

Step 1: Run the given server-client example as shown below (see terminal 1 and terminal 3 outputs). Start a server (terminal 1), and then start a client (terminal 3). Once the program runs, make sure you understand what the program is doing, its networking API calls.

Step 2: Now together with a working server-client example, we run a tcpdump example (terminal 2). In the first terminal window, you should start the server:

Terminal 1

```
# start the server
vuser@vm:~/anp_netskeleton/$ ./build/anp_server -a 192.168.56.101 -p 33320 -c 1
[server] working with the following IP: 192.168.56.101 and port 33320 (config: 1)
Socket successfully created, fd = 3
Socket successfully bind'ed
Server listening.
# ← server will wait here, until a new connection has arrived...

new incoming connection from 192.168.56.101
    [receive loop] 4096 bytes, looping again, so_far 4096 target 4096
OK: buffer received ok, pattern match : < OK, matched >
    [send loop] 4096 bytes, looping again, so_far 4096 target 4096
OK: buffer tx backed
ret from the recv is 0 errno 0
OK: server and client sockets closed
vuser@vm:~/anp_netskeleton/$
```

In a second terminal window, you should start the tcpdump program (*always use “-nN” flags with the tcpdump - find out what these flags do?*):

Terminal 2

```
# start the packet capture
vuser@vm:~$ sudo tcpdump -nN --absolute-tcp-sequence-numbers -v -i lo port 33320 -B
$((1024*32))
tcpdump: listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
# ← tcpdump will wait here and when you start the client in the 3rd window, the following
output will appear ...which shows individual packet exchange.

10:31:32.357226 IP (tos 0x0, ttl 64, id 56573, offset 0, flags [DF], proto TCP (6), length 60)
    192.168.56.101.53274 > 192.168.56.101.33320: Flags [S], cksum 0xf249 (incorrect -> 0xba6b), seq
1711795356, win 64240, options [mss 1460,sackOK,TS val 4281793641 ecr 0,nop,wscale 7], length 0
10:31:32.357246 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 60)
    192.168.56.101.33320 > 192.168.56.101.53274: Flags [S.], cksum 0xf249 (incorrect -> 0x7d92), seq
4258217919, ack 1711795357, win 65160, options [mss 1460,sackOK,TS val 4281793641 ecr
4281793641,nop,wscale 7], length 0
10:31:32.357258 IP (tos 0x0, ttl 64, id 56574, offset 0, flags [DF], proto TCP (6), length 52)
    192.168.56.101.53274 > 192.168.56.101.33320: Flags [.], cksum 0xf241 (incorrect -> 0xa8f1), ack
4258217920, win 502, options [nop,nop,TS val 4281793641 ecr 4281793641], length 0
10:31:32.357360 IP (tos 0x0, ttl 64, id 56575, offset 0, flags [DF], proto TCP (6), length 1500)
    192.168.56.101.53274 > 192.168.56.101.33320: Flags [.], cksum 0xf7e9 (incorrect -> 0xb388), seq
1711795357:1711796805, ack 4258217920, win 502, options [nop,nop,TS val 4281793641 ecr 4281793641],
length 1448
10:31:32.357370 IP (tos 0x0, ttl 64, id 56576, offset 0, flags [DF], proto TCP (6), length 1500)
    192.168.56.101.53274 > 192.168.56.101.33320: Flags [.], cksum 0xf7e9 (incorrect -> 0x81b4), seq
1711796805:1711798253, ack 4258217920, win 502, options [nop,nop,TS val 4281793641 ecr 4281793641],
length 1448
10:31:32.357376 IP (tos 0x0, ttl 64, id 56577, offset 0, flags [DF], proto TCP (6), length 1252)
```

```

192.168.56.101.53274 > 192.168.56.101.33320: Flags [P.], cksum 0xf6f1 (incorrect -> 0xa09b), seq
1711798253:1711799453, ack 4258217920, win 502, options [nop,nop,TS val 4281793641 ecr 4281793641],
length 1200
10:31:32.357380 IP (tos 0x0, ttl 64, id 15823, offset 0, flags [DF], proto TCP (6), length 52)
192.168.56.101.33320 > 192.168.56.101.53274: Flags [.), cksum 0xf241 (incorrect -> 0xa34a), ack
1711796805, win 501, options [nop,nop,TS val 4281793641 ecr 4281793641], length 0
10:31:32.357381 IP (tos 0x0, ttl 64, id 15824, offset 0, flags [DF], proto TCP (6), length 52)
192.168.56.101.33320 > 192.168.56.101.53274: Flags [.), cksum 0xf241 (incorrect -> 0x9da7), ack
1711798253, win 496, options [nop,nop,TS val 4281793641 ecr 4281793641], length 0
10:31:32.357382 IP (tos 0x0, ttl 64, id 15825, offset 0, flags [DF], proto TCP (6), length 52)
192.168.56.101.33320 > 192.168.56.101.53274: Flags [.), cksum 0xf241 (incorrect -> 0x98fc), ack
1711799453, win 491, options [nop,nop,TS val 4281793641 ecr 4281793641], length 0
10:31:32.357420 IP (tos 0x0, ttl 64, id 15826, offset 0, flags [DF], proto TCP (6), length 1500)
192.168.56.101.33320 > 192.168.56.101.53274: Flags [.), cksum 0xf7e9 (incorrect -> 0xa389), seq
4258217920:4258219368, ack 1711799453, win 501, options [nop,nop,TS val 4281793641 ecr 4281793641],
length 1448
10:31:32.357422 IP (tos 0x0, ttl 64, id 15827, offset 0, flags [DF], proto TCP (6), length 1500)
192.168.56.101.33320 > 192.168.56.101.53274: Flags [.), cksum 0xf7e9 (incorrect -> 0x71b5), seq
4258219368:4258220816, ack 1711799453, win 501, options [nop,nop,TS val 4281793641 ecr 4281793641],
length 1448
10:31:32.357423 IP (tos 0x0, ttl 64, id 15828, offset 0, flags [DF], proto TCP (6), length 1252)
192.168.56.101.33320 > 192.168.56.101.53274: Flags [P.], cksum 0xf6f1 (incorrect -> 0x909c), seq
4258220816:4258222016, ack 1711799453, win 501, options [nop,nop,TS val 4281793641 ecr 4281793641],
length 1200
10:31:32.357435 IP (tos 0x0, ttl 64, id 56578, offset 0, flags [DF], proto TCP (6), length 52)
192.168.56.101.53274 > 192.168.56.101.33320: Flags [.), cksum 0xf241 (incorrect -> 0x8911), ack
4258222016, win 470, options [nop,nop,TS val 4281793641 ecr 4281793641], length 0
10:31:37.357962 IP (tos 0x0, ttl 64, id 56579, offset 0, flags [DF], proto TCP (6), length 52)
192.168.56.101.53274 > 192.168.56.101.33320: Flags [F.), cksum 0xf241 (incorrect -> 0x7568), seq
1711799453, ack 4258222016, win 501, options [nop,nop,TS val 4281798642 ecr 4281793641], length 0
10:31:37.358026 IP (tos 0x0, ttl 64, id 15829, offset 0, flags [DF], proto TCP (6), length 52)
192.168.56.101.33320 > 192.168.56.101.53274: Flags [F.), cksum 0xf241 (incorrect -> 0x61de), seq
4258222016, ack 1711799454, win 501, options [nop,nop,TS val 4281798642 ecr 4281798642], length 0
10:31:37.358049 IP (tos 0x0, ttl 64, id 56580, offset 0, flags [DF], proto TCP (6), length 52)
192.168.56.101.53274 > 192.168.56.101.33320: Flags [.), cksum 0xf241 (incorrect -> 0x61de), ack
4258222017, win 501, options [nop,nop,TS val 4281798642 ecr 4281798642], length 0
^C
16 packets captured
32 packets received by filter
0 packets dropped by kernel
user@vm:~/anp_netskeleton/$

```

In a third terminal window, you can start the client:

Terminal 3:

```

# start the client for transfer
vuser@vm:~/anp_netskeleton/$ ./build/anp_client -a 192.168.56.101 -p 33320 -c 1
[client] working with the following IP: 192.168.56.101 and port 33320 (config: 1)
OK: socket created, fd is 3
OK: connected to the server at 192.168.56.101
[send loop] 4096 bytes, looping again, so_far 4096 target 4096
OK: buffer sent successfully
OK: waiting to receive data
[receive loop] 4096 bytes, looping again, so_far 4096 target 4096
Results of pattern matching: < OK, matched >
A 5 sec wait before calling close
OK: shutdown was fine. Good bye!
Time for the test is (microseconds): 1376
vuser@vm:~/anp_netskeleton/$

```

There are a few things to notice and test in the above run:

1. **Checksum on a local machine:** When you run your code on a loopback address, Linux kernel does not do checksum and hence you see incorrect checksum fields. <https://www.spinics.net/lists/netdev/msg170879.html>. However, you need a working checksum example in order to pass this assignment. **A wrong checksum calculation will be considered a valid implementation.** In order to see a working example, put the server outside the virtual machine and start on the host machine as shown below:

```
# start the server on outside the VM on a host with a host's IP address
user@host:~/anp_netskeleton/$./build/anp_server -a 145.108.76.180 -p 33320 -c 1
```

At this stage, when you capture packet from the client which is running inside the virtual machine, it will show something like:

```
# example capture of a tcpdump with valid checksum calculation
10:37:01.303528 IP (tos 0x0, ttl 64, id 35237, offset 0, flags [DF], proto TCP (6), length 60)
    192.168.56.101.42570 > 145.108.76.180.33320: Flags [S], cksum 0xcb51 (correct), seq 1646221954, win 64240, options [mss 1460,sackOK,TS val 2506436629 ecr 0,nop,wscale 7], length 0
10:37:01.303682 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 60)
    145.108.76.180.33320 > 192.168.56.101.42570: Flags [S.], cksum 0x1daa (correct), seq 3338265063, ack 1646221955, win 65160, options [mss 1460,sackOK,TS val 2227600470 ecr 2506436629,nop,wscale 7], length 0
10:37:01.303724 IP (tos 0x0, ttl 64, id 35238, offset 0, flags [DF], proto TCP (6), length 52)
    192.168.56.101.42570 > 145.108.76.180.33320: Flags [.], cksum 0x4909 (correct), ack 3338265064, win 502, options [nop,nop,TS val 2506436629 ecr 2227600470], length 0
10:37:01.303795 IP (tos 0x0, ttl 64, id 35239, offset 0, flags [DF], proto TCP (6), length 1500)
    192.168.56.101.42570 > 145.108.76.180.33320: Flags [.], cksum 0x53a0 (correct), seq 1646221955:1646223403, ack 3338265064, win 502, options [nop,nop,TS val 2506436629 ecr 2227600470], length 1448
10:37:01.303807 IP (tos 0x0, ttl 64, id 35240, offset 0, flags [DF], proto TCP (6), length 1500)
    192.168.56.101.42570 > 145.108.76.180.33320: Flags [.], cksum 0x21cc (correct), seq 1646223403:1646224851, ack 3338265064, win 502, options [nop,nop,TS val 2506436629 ecr 2227600470], length 1448
10:37:01.303808 IP (tos 0x0, ttl 64, id 35241, offset 0, flags [DF], proto TCP (6), length 1252)
    192.168.56.101.42570 > 145.108.76.180.33320: Flags [P.], cksum 0x40b3 (correct), seq 1646224851:1646226051, ack 3338265064, win 502, options [nop,nop,TS val 2506436629 ecr 2227600470], length 1200
10:37:01.303918 IP (tos 0x0, ttl 64, id 38699, offset 0, flags [DF], proto TCP (6), length 52)
    145.108.76.180.33320 > 192.168.56.101.42570: Flags [.], cksum 0x4362 (correct), ack 1646223403, win 501, options [nop,nop,TS val 2227600470 ecr 2506436629], length 0
10:37:01.303919 IP (tos 0x0, ttl 64, id 38700, offset 0, flags [DF], proto TCP (6), length 52)
    145.108.76.180.33320 > 192.168.56.101.42570: Flags [.], cksum 0x3dbf (correct), ack 1646224851, win 496, options [nop,nop,TS val 2227600470 ecr 2506436629], length 0
10:37:01.303920 IP (tos 0x0, ttl 64, id 38701, offset 0, flags [DF], proto TCP (6), length 52)
    145.108.76.180.33320 > 192.168.56.101.42570: Flags [.], cksum 0x3915 (correct), ack 1646226051, win 490, options [nop,nop,TS val 2227600470 ecr 2506436629], length 0
10:37:01.304065 IP (tos 0x0, ttl 64, id 38702, offset 0, flags [DF], proto TCP (6), length
```

```

1500)
  145.108.76.180.33320 > 192.168.56.101.42570: Flags [.], cksum 0x43a1 (correct), seq
3338265064:3338266512, ack 1646226051, win 501, options [nop,nop,TS val 2227600470 ecr
2506436629], length 1448
10:37:01.304076 IP (tos 0x0, ttl 64, id 38703, offset 0, flags [DF], proto TCP (6), length
1500)
  145.108.76.180.33320 > 192.168.56.101.42570: Flags [.], cksum 0x11cd (correct), seq
3338266512:3338267960, ack 1646226051, win 501, options [nop,nop,TS val 2227600470 ecr
2506436629], length 1448
10:37:01.304078 IP (tos 0x0, ttl 64, id 38704, offset 0, flags [DF], proto TCP (6), length
1252)
  145.108.76.180.33320 > 192.168.56.101.42570: Flags [P.], cksum 0x30b4 (correct), seq
3338267960:3338269160, ack 1646226051, win 501, options [nop,nop,TS val 2227600470 ecr
2506436629], length 1200
10:37:01.304106 IP (tos 0x0, ttl 64, id 35242, offset 0, flags [DF], proto TCP (6), length
52)
  192.168.56.101.42570 > 145.108.76.180.33320: Flags [.], cksum 0x3362 (correct), ack
3338266512, win 501, options [nop,nop,TS val 2506436629 ecr 2227600470], length 0
10:37:01.304126 IP (tos 0x0, ttl 64, id 35243, offset 0, flags [DF], proto TCP (6), length
52)
  192.168.56.101.42570 > 145.108.76.180.33320: Flags [.], cksum 0x2dbf (correct), ack
3338267960, win 496, options [nop,nop,TS val 2506436629 ecr 2227600470], length 0
10:37:01.304137 IP (tos 0x0, ttl 64, id 35244, offset 0, flags [DF], proto TCP (6), length
52)
  192.168.56.101.42570 > 145.108.76.180.33320: Flags [.], cksum 0x2916 (correct), ack
3338269160, win 489, options [nop,nop,TS val 2506436629 ecr 2227600470], length 0
10:37:06.305237 IP (tos 0x0, ttl 64, id 35245, offset 0, flags [DF], proto TCP (6), length
52)
  192.168.56.101.42570 > 145.108.76.180.33320: Flags [F.], cksum 0x1580 (correct), seq
1646226051, ack 3338269160, win 501, options [nop,nop,TS val 2506441630 ecr 2227600470],
length 0
10:37:06.306076 IP (tos 0x0, ttl 64, id 38705, offset 0, flags [DF], proto TCP (6), length
52)
  145.108.76.180.33320 > 192.168.56.101.42570: Flags [F.], cksum 0x01f5 (correct), seq
3338269160, ack 1646226052, win 501, options [nop,nop,TS val 2227605472 ecr 2506441630],
length 0
10:37:06.306225 IP (tos 0x0, ttl 64, id 35246, offset 0, flags [DF], proto TCP (6), length
52)
  192.168.56.101.42570 > 145.108.76.180.33320: Flags [.], cksum 0x01f4 (correct), ack
3338269161, win 501, options [nop,nop,TS val 2506441631 ecr 2227605472], length 0

```

2. **Make sure to have the right NIC settings:** Make sure you have checksum offloading disabled on the NICs (sudo ethtool -K enp0s3 tx off rx off). This should have been done automatically when you set up forwarding rules with the TAP device. Cross check.
3. **Pay attention to tcpdump config:** Pay attention to the -NN field, -B, which NIC, and which port, which protocol filter are you applying to tcpdump.

Once you have the given framework and server-client example working, now it is time to start coding your own TCP implementation. You can hijack the networking call from the client side code as:

```

# start the server on outside the VM on a host with a host's IP address
vuser@vm:~/anp_netskeleton/$sudo ./bin/sh-hack-anp.sh ./build/anp_client -a 145.108.76.180
-p 33320 -c 1
+ prog=./build/anp_client

```

```

+ shift
+ LD_PRELOAD=/usr/local/lib/libanpnetstack.so
+ ./build/anp_client -a 145.108.76.180 -p 33320 -c 1
Hello there, I am ANP networking stack!
tap device OK, tap0
Executing : ip link set dev tap0 up
OK: device should be up now, 10.0.0.0/24
Executing : ip route add dev tap0 10.0.0.0/24
OK: setting the device route, 10.0.0.0/24
Executing : ip address add dev tap0 local 10.0.0.5
OK: setting the device address 10.0.0.5
GXXX 0.0.0.0
GXXX 0.0.0.0
GXXX 10.0.0.5
[client] working with the following IP: 145.108.76.180 and port 33320 (config: 1)
supported socket domain 2 type 1 and protocol 0
anp_client: /home/atr/anp_netskeleton/src/anpwrapper.c:65: socket: Assertion `ret == 0'
failed.
./bin/sh-hack-anp.sh: line 9: 8449 Aborted                  (core dumped)
LD_PRELOAD="/usr/local/lib/libanpnetstack.so" "$prog" "$@"

```

This is where you start to code your project for the milestone. This milestone is about (i) allocating a socket - implement `socket()` call; (ii) establishing a three-way connection handshake - implement `connect()`; and (iii) sending data bytes to the server - implement `send()` call. At the end of the milestone 2, you will have a working IP, ARP, and ICMP implementation. The TCP protocol runs on top of IP.

The basic TCP functionality is defined in the famous **RFC 793** <https://tools.ietf.org/html/rfc793>. Section 3 (Protocol Specification) is the most important section for you to read and understand. As a guideline, for any input TCP packet you can follow processing as defined in the section 3.9, <https://tools.ietf.org/html/rfc793#section-3.9>.

How to start and approach the milestone:

1. Implement the `socket()`, `connect()`, and `send()` calls. See the `anpwrapper.c` file.
2. **Socket call:**
 - a. Understand what this call does? Read the man page: <https://man7.org/linux/man-pages/man2/socket.2.html>
 - b. What parameters does it take, and what value does it return?
 - c. Can you implement any or all `socket()` calls? How do you know which `socket()` calls you need to implement, and let others go to the Linux kernel? Ensure that you only override networking calls for the sockets (i.e., file descriptors) that your ANP stack allocated. Not any other. We target to implement only `TCP_STREAM` on IPv4 socket calls.
 - d. What is a file descriptor? Understand that all sockets are also file descriptors. You need to manage a socket file descriptor space. For convenience you can pick a large number like one million and generate file descriptors from then onwards.
 - e. What can you allocate and return?
3. **Connect call**
 - a. Understand what this call does? Read the man page: <https://man7.org/linux/man-pages/man2/connect.2.html>

- b. What parameters does it take, and what value does it return?
 - c. Can you implement any or all connect() calls? How do you know which connect() calls you need to implement, and let others go to the Linux kernel? Ensure that you only override networking calls for the sockets (i.e., file descriptors) that your ANP stack allocated. Not any other.
 - d. You need to send out a TCP packet with a connection request. Define a TCP packet header with all the fields and make sure that the header size is correct.
 - e. Think about what values you need to populate in the fields. Are you packing the TCP header properly in the SU buffer? Does your expectation match what you capture in the tcpdump?
 - f. Think what happens if this packet is lost? How would you know a packet is lost? Like anything in life, if you fail, try again. How many times should you try? Is there a time gap in between the trails?
 - g. With the right timeout (see *timer.h* file) you should try to send the packet again (for a certain number of retries). Always check the return code for function calls.
 - h. How and when do you know if your connect() call works? What do you expect to get from the server?
 - i. What do you need to do in a three-way handshake? Have you generated the acknowledgements for the packets that you received?
 - j. How do you calculate checksum?
 - k. When do you return "all good" (i.e., a zero return value) to the user code which called the connect() call.
 - l. Have you implemented the TCP state machine? What TCP state transitions are needed for the connect call?
- 4. Send call():**
- a. Understand what this call does? Read the man page: <https://man7.org/linux/man-pages/man2/send.2.html>
 - b. What parameters does it take, and what value does it return?
 - c. Can you implement any or all send() calls? How do you know which send() calls you need to implement, and let others go to the Linux kernel?
 - d. How do you construct a packet with user data?
 - e. How do you calculate checksum?
 - f. How do you identify if a send() packet is lost?
 - g. Have you generated the acknowledgements for the packets that you received?
 - h. How and when do you know if your send() call works? What do you expect to get from the server?
 - i. Have you implemented the TCP state machine? What TCP state transitions are needed for the send call?
- 5. Generate the tcpdump profile of packets exchanged (you can filter your application specific packets by combining the NIC name and port number in the tcpdump command). You will show this profile during the interview and explain what is happening.
 - 6. You are not expected to support urgent pointers or any TCP options.

- At the end of this assignment, your TCP state machine should be in an ESTABLISHED state, you sent the complete data buffer and processed incoming ACKs, and are ready to receive data from the server (milestone 4).

Hint for doing TCP checksum generation for any outgoing packet (including connection): File utilities.c defines the following function:

```
int do_tcp_csum(uint8_t *data, int length, uint16_t protocol, uint32_t saddr, uint32_t daddr)
```

This can be used to calculate the checksum of your outgoing TCP packets. The arguments of this function are as follows:

uint8_t data	The starting address of the TCP header
int length	The length of the TCP packet (header + data if present)
uint16_t protocol	must be set to IPP_TCP
uint32_t saddr	source address (host byte-order format)
uint32_t daddr	destination address (host byte-order format)

It is important to note that when you calculate the checksum all other header fields must be set to their correct value and they must already be in network byte-order format, as such, the checksum is effectively the last field of the header to set before sending the packet.

Regarding the source address, your application-private IP is 10.0.0.4 and MAC address is "de:ad:be:ef:aa:aa" defined in config.h file. This IP and MAC address is used for the packet when they leave your shared library and go to the TAP device. At this moment their goal is to get to the TAP device because that is the only connection from the application-private NIC to the outside world. The server destination IP is given in the connect call, and the MAC is the MAC address of the gateway node (10.0.0.5) which the arp stack will resolve using the ARP protocol (just like in milestone 1).

Unexpected reset packets: When developing the connect logic it might be the case that you get unexpected packets or reset request packets from the remote server node after you have sent the initial SYN packet and then crashed (your library code crashed). This is normal, as the peer host (the host where the server is running) does not know what happened to a connection for which it has received the initial SYN packet, and will continue to ask and wait for it for a while (connection timeout). Hence, when the next time you try to connect again to the same IP + port pair, the server host will try to reset the connection as it is still in the old connection state. In case you suspect something like this is happening, change the port number and use a new port number. This way you can continue debugging your code without having to wait for the timeout duration after which the original IP port pair becomes available again.

Seeing an unexpected number of packets with tcpdump? Make sure to disable offload settings on your NIC

```
vm@vhost:$ $ ifconfig lo mtu 1500
vm@vhost:$ ifconfig enp0s3 mtu 1500
vm@vhost:$ ethtool -K enp0s3 tso off lro off gro off gso off
vm@vhost:$ ethtool -K lo tso off lro off gro off gso off
```

Do's:

1. At the end of the day, the ANP is still a userspace program, hence all libraries, facilities, and infrastructure for building and debugging programs are open for you. Use them. Any Linux debugger should be able to help you.
2. Learn how to use network debugging tools like **netcat**, **netstat**, or **nmap** to help you debug your networking behavior.
3. Think how you can narrow down your complexity and debugging space. Is the problem on the sending side or receiving side?
4. Put prints and generate logs and events. Spend time looking into the framework itself. It is given to you for a reason for you to see how a networking stack looks like.
5. Think and implement for things that can go wrong: packet lost, duplicate packets, corrupted packets, re-ordered packets! Your code is judged on the error handling completeness.
6. If you finish early with this assignment then proceed to Milestone 4.

Don'ts:

1. Do not use busy loops (`while (something);`) and sleep syscall (`sleep(5)`) for an unspecified or random amount of time. Use condition variables from *pthread* to trigger an action on a certain event and *time.h* for timeouts.
2. Do not hardcode your TCP implementation, create a proper state machine and check for legal and illegal transitions and allowed actions.
3. **IMPORTANT: Do not assume any packet ordering from the network**, for example after sending the data packet the next incoming packet *_must_* be an acknowledgement. So a code sequence such as (i) send one data packet and loop to receive the next packet; (ii) process the incoming packet as the TCP ack packet (no checks); (iii) send the next data packet - is not appropriate code. ***In the interview, the more general your code is (checks, error handling), the better grade you will get.***

Evaluation of Milestone 3 is done as a short interview. More details will be given in the class as we approach the evaluation week.

IMPORTANT: this is a heavy milestone with a lot of complexity and coding. There is a reason it has more time than the other assignments with more than one lab session in between. Start early and ask questions. We will not be able to help one day before the deadline. There is a lot that can go wrong here.

Milestone summary: What needs to be done:

1. Setup and run the server-client example program with the Linux netstack.
2. Make sure you can capture the packets and understand what is happening with tcpdump.
3. Now run the client side code, with your ANP netstack.
4. Implement socket, connect, and send calls in the libanp framework - make sure that the client side code can be run multiple times (not just once), until the point the recv() call fails.
5. Understand what you can implement and what is now allowed. Do not make network event ordering assumptions regarding which packet will arrive or send when. When in doubt, ask us.
6. Prepare 1-2 slides to give an “graphical” visualization of the code you implement, mostly focusing on choice of data structure.
7. Zip and the source code with the complete framework, and upload on Canvas within the deadline.
8. **Make sure you include everything with a running code. If we cannot compile your code, you get a zero. You will not get an extension to fix your uploading mistakes.**
9. **Prepare for the M3 interview and show up.**

4.4 Milestone 4: *Careful With That Data, Eugene*

Type: group (2-3)

Maximum points: 10

Where and what: (1) Take the canvas quiz and (2) code upload on canvas (zip file)

Deadline: see the course canvas page

The fourth milestone is about completing the recv() and close() calls. The client receives a buffer and matches a pattern on them that must match. If successful, then the client will close the connection after waiting for 5 seconds. Most of the details of this milestone are the same as in the last one.

How to start and approach the milestone:

1. Implement and complete the `recv()`, and `close()` calls. See the anpwrapper.c file.
2. **recv() call:**
 - a. Understand what this call does? Read the man page: <https://man7.org/linux/man-pages/man2/recvmsg.2.html>
 - b. What parameters does it take, and what value does it return?
 - c. Can you implement any or all recv() calls? How do you know which recv() calls you need to implement, and let others go to the Linux kernel?
 - d. How do you process an incoming TCP packet with data? Is it a valid packet? Do the checksum match?
 - e. Have you generated the acknowledgements for the packets that you received? What happens if the acknowledgment is lost?
 - f. How and when do you know if your recv() call works? Do you expect anything from the server side?
 - g. When do you copy data to the user buffer? When do you generate an acknowledgement?

- h. Have you implemented the TCP state machine? What TCP state transitions are needed for the `recv` call?
- 3. **close() call:**
 - a. Understand what this call does? Read the man page: <https://man7.org/linux/man-pages/man2/close.2.html>
 - b. What parameters does it take, and what value does it return?
 - c. Can you implement any or all `close()` calls? How do you know which `close()` calls you need to implement, and let others go to the Linux kernel?
 - d. What needs to be done for the TCP state machine processing?
 - e. How do you prepare and send a TCP close connection packet?
 - f. How and when do you know if your `close()` call works? Do you expect anything from the server side?
- 4. Generate a complete `tcpdump` trace for the packet exchanges that happen between the TCP server client program during the `connect`, `send`, `recv`, and `close` calls. Upload this TCP dump trace with an explanation. We will ask to show this trace in the final Milestone 5 interview.
- 5. Compare the trace generated by your netstack with the default Linux trace, write a short explanation of your analysis (max 2 pages) of the `tcpdump`, and upload this together with the code in Canvas.

Valid assumptions to make:

- 1. You are not supposed to implement the congestion control mechanism. However you must ensure that you do not run over a receiver's advertised receive window. If any more data is requested to be sent, you must suspend the calling thread and wait until more acknowledgements are received.
- 2. You do not have to implement out-of-order receives. You can drop such segments.
- 3. You do not have to implement any advanced acknowledgement mechanisms (e.g., SACK, NACK, delayed ACKs, etc.). However, these can be bonus assignments.
- 4. You can skip the TCP checksum match on the receive path. However, you must implement the correct TCP checksum logic on the send path.
- 5. For TCP 4-way close handshake please see <https://tools.ietf.org/html/rfc793#section-3.5> (you are supposed to implement the "Case 1: Local user initiates the close", which is the TCP client side in the example code).
- 6. TCP closing routine: <https://community.apigee.com/articles/7970/tcp-states-explained.html>

You should be able to demonstrate and explain the TCP closing mechanism using `tcpdump` packet trace. Here is an example using the default Linux in-kernel netstack (pay attention to the flags field):

```
14:03:41.481575 IP 192.168.1.161.41730 > atr-XPS-13.44441: Flags [F.], seq 17, ack 17, win 229, options [nop,nop,TS val 2019541374 ecr 2799286268], length 0
14:03:41.481829 IP atr-XPS-13.44441 > 192.168.1.161.41730: Flags [F.], seq 17, ack 18, win 509, options [nop,nop,TS val 2799291272 ecr 2019541374], length 0
14:03:41.482162 IP 192.168.1.161.41730 > atr-XPS-13.44441: Flags [.], ack 18, win 229, options [nop,nop,TS val 2019541375 ecr 2799291272], length 0
```

At this stage, the complete TCP client-server program should be running using your ANP stack implementation.

Milestone summary: What needs to be done:

1. Complete the `recv()` and `close()` calls.
2. Generate a complete `tcpdump` trace of packet exchanges
3. Write a short paragraph (PDF) of explanation what is happening in the complete packet exchange
4. Take the canvas quiz.
5. Zip and the source code with the complete framework, `tcpdump`, explanation PDF, and upload on Canvas within the deadline.
6. **Make sure you include everything with a running code. If we cannot compile your code, you get a zero. You will not get an extension to fix your uploading mistakes.**

4.5 Milestone 5: Another Graph in the Wall

Type: group (2-3)

Maximum points: 10

Where and what: (1) Code upload on canvas (zip file) and (2) show up for the in-class interview

Deadline: see the course Canvas page

In this milestone you are going to perform a measurement study to quantify the performance of the network stack you just created, in comparison to the Linux kernel network stack. At the end, you are supposed to plot your measurement results and draw conclusions from your plots.

Specifically, you are supposed to transfer a certain amount of data using your network stack and the Linux kernel network stack and measure the end-to-end latency of the data transfers. Please perform this measurement with 4KB, 32KB, and 1MB of data to transfer respectively and repeat each experiment **ten times**. The three buffer sizes can be selected with the `-c` flag in the program (`-c 1 = 4KB`, `-c 2 = 32KB`, and `-c 3 = 1MB`). The time it takes to run the program is printed at the end of the program run at the client side. Pay attention to the output there.

Once you obtain the latency numbers, please plot these numbers with box plots, one plot for each data transfer. If you are not familiar with box plots, please refer to this page: https://en.wikipedia.org/wiki/Box_plot. On each plot, there should be two bars, for your network stack and the Linux kernel network stack, respectively. ***Please calculate the numbers for the box plots by writing your own code, not with existing libraries.*** Learn how to calculate quartile numbers for a given set of measurement values (here: 10 samples from 10 runs).

Please prepare one slide with the three plots and add a couple of bullet points summarizing your findings, key observations, and explanation of, for example, why your stack is slow or fast. During the interview, you are expected to show the slide, and explain how the data was collected and processed, what observations you have, and how you reason about these observations.

The M5 interview will be about the whole project.

Milestone summary: What needs to be done:

1. Perform a measurement study to quantify the data transfer latency (for 4KB, 32KB, and 1MB of data) with your network stack in comparison with the Linux kernel network stack. Repeat each data transfer ten times.
 - a. Make sure your code can run with -c 1 , -c 2 and -c 3 flags.
 - b. Collect runtimes from 10 runs from your code for above mentioned -c (three) configurations.
 - c. Learn how to calculate and plot a box plot with the data collected. Process the measured numbers and create three box plots for three configurations respectively.
 - d. Repeat steps 1-3 for the Linux network stack.
 - e. Plot Linux measurements next to your own stack's measurements.
2. Prepare one slide with the three box plots and a summary of your findings as bullet points.
3. Zip and the source code with the complete framework, the slides, and upload on Canvas within the deadline.
4. **Make sure you include everything with a running code. If we cannot compile your code, you get a zero. You will not get an extension to fix your uploading mistakes.**
5. **Prepare for the M5 interview (about the whole project) and show up.**

5. Bonus Ideas and Timeline

Each of these bonus ideas is worth 10 points. There are no partial marks for the bonus. You need to implement the bonus, and demonstrate that it works by implementing additional code that exercises your bonus feature implementation, and show **conclusively** that your implementation works.

- Implement one of the TCP acknowledge schemes: NACK, SACK, or delayed ACK and quantify their impact on the performance of your networking stack.
- Implement IP fragmentation and reassembly logic.
- Multi-tenancy/multi-threading support in your stack
- Multi-core scalability, implement multiple rx and tx threads per core, support asynchronous sending/receiving
- Socket NON_BLOCKING features with select/poll/epoll call implementations, make a small test program to show they work
- Any ideas of your choosing, please consult with us.

Bonus++ : There is a special bonus assignment with which you can get additional 0.5 points your final grade. The goal is to run a browser (firefox, chrome) on your networking stack and submit the final milestone 5 on canvas. You should be able to transparently hijack your browser's networking code with your networking stack implementation from milestone 4. Firefox uses more calls like epoll, select, with different flags to set the behavior of the socket, multithreading, etc.

When to demonstrate the bonus work: The deadline for showing us the bonus work is Milestone 5 interview. During that session you must claim all your bonus marks. **You must inform us by the milestone 3 interview day that you plan to do bonus work, which feature you are implementing, and how do you plan to demonstrate that they work. You need to get an explicit OK from us for the bonus.** If you did not get us then your bonus implementation is not valid. *On the spot bonus will not be considered for grading.* You will either get the full marks, or none for the bonus.

Appendix

A. Important RFCs

- TCP, <https://tools.ietf.org/html/rfc793>
- TCP MSS (segment size), <https://tools.ietf.org/html/rfc879>
- ICMP, <https://tools.ietf.org/html/rfc792>