

# UTM Network Programming course for FAF-23 (Autumn 2025)

## Books

I recommend reading “Computer Networking: a Top Down Approach”, skipping chapter 1. In chapter 2 you will learn everything you need for lab 1. You can find a copy in [this folder](#).

When learning about concurrency, beware that there are many conflicting definitions floating around, some of which are misleading and might cause you unnecessary suffering. I recommend adopting the definitions from [this glossary](#) and [this paper](#). [This article](#) from an MIT course is a good introduction to the topic.

## Calendar

	Started	Deadline	Duration
Lab 1	Sep 30, 2025	Oct 14, 2025	2 weeks
Lab 2	Oct 14, 2025	Oct 21, 2025	1 weeks
Mid 1	Oct 27, 2025		
Lab 3	Oct 28, 2025	Nov 11, 2025	2 weeks
Lab 4	Nov 11, 2025	Nov 25, 2025	2 weeks

## Labs

The PR labs will be interesting because you will write programs to network with your colleagues (e.g. query your friend’s HTTP server using your own client, host an online game server and play with your friends, etc.)

Rules:

- You must use Docker Compose
- You must use Python

## How to present

Upload your project to GitHub and send me the URL in an email with the subject “PR lab 1” ([artiom.balan@isa.utm.md](mailto:artiom.balan@isa.utm.md)). If you will be presenting later than the deadline, include the last commit hash so I can verify the time of the commit.

Include a short report in the repository (markdown or a link to a Google doc). The report will be like a demo but with screenshots, so just include screenshots of everything you would need to show in the demo, each described with a short sentence: the commands you run, the outputs you see (browser, terminal). The report should prove that you satisfied all the requirements of the lab.

You will present to me during seminars, defend your code, and answer a few theoretical questions. I will give priority to those who sent me an email first.

## Docker

You must use Docker Compose for all your laboratory works so that I can run them on my computer.

The official [Docker 101 tutorial](#) is a good resource.

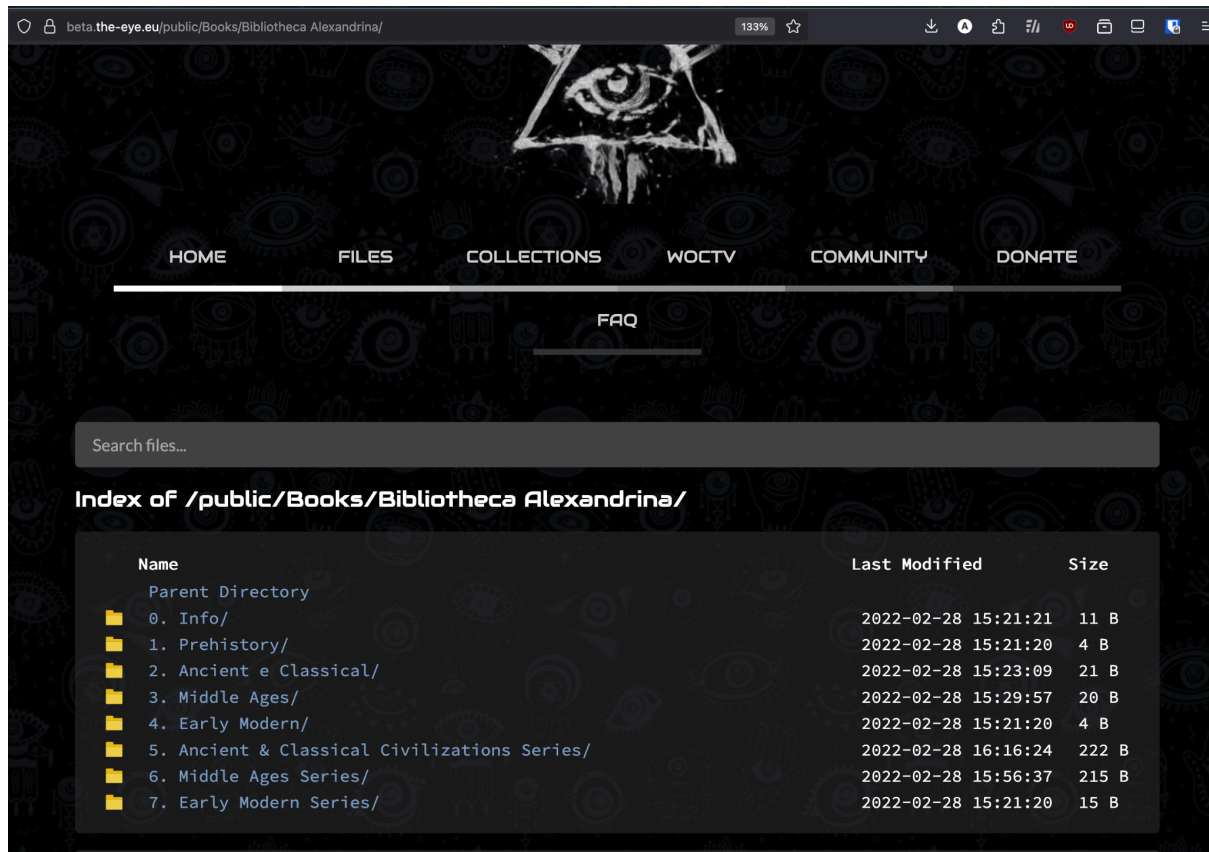
## Lab 1: HTTP file server with TCP sockets

The questions will be based on these resources:

- Chapter 2 from “Computer Networking 8th edition” (you can skip 2.3-2.6)
- [This Python HOWTO article](#)
- (Optional) [MIT class reading on Sockets & Networking](#)

You can practice on the questions from the end of the chapter.

You will develop a simple HTTP file server like Python’s `http.server`. You will use it to build a website that lets your friends browse your collection of PDFs, like [this one](#):



It will handle one HTTP request at a time. The server program should take the directory to be served as a command-line argument.

Your web server should accept and parse the HTTP request, read the requested file from the directory, create an HTTP response message consisting of the requested file preceded by header lines, and then send the response directly to the client. If the requested file is not present in the server (or has an unknown extension), the server should send an HTTP "404 Not Found" message back to the client. **Update:** your server should handle HTML, PNG and PDF file types.

Prepare a directory with content to be served. At the very least, it should contain an HTML file, a PNG image and a few PDF files. Reference the image in the HTML file (using `<img>`).

To get a 10, you need to do the following tasks as well:

1. Implement an HTTP client for your server - **2 points**

**Update:** It will be a python script that takes as command-line arguments a URL and a directory to save files in, and acts depending on the file type:

- HTML (page, directory listing): print the body of the response as-is
- PNG, PDF: save the file in the specified directory

Use the following format for the command arguments:  
 client.py server\_host server\_port url\_path directory

2. Make the server work with nested directories - **2 points:**

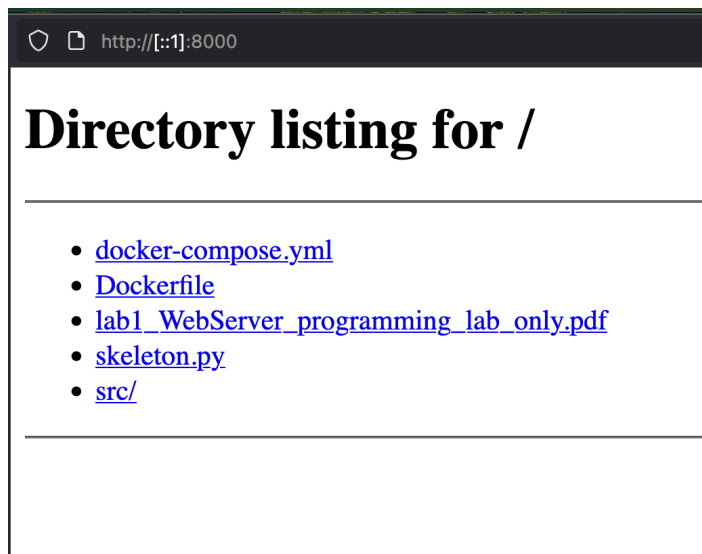
If a directory path is requested, respond with a directory listing (a generated HTML page that displays the contents of the directory, with hyperlinks).

Create a subdirectory in your content directory with a few PDF/PNG files inside it.

3. Browse the books on your friend's server (local network), have fun - **1 point**

You can use your client to download a book from your friend's server right into your own website directory.

What the directory listing can look like (python's http.server response shown here):



Report contents:

- The contents of your source directory
- The docker compose file (and Dockerfile if you use one)
- show how you start the container
- Command that runs the server inside the container, with a directory as an argument
- The contents of the served directory
- Requests of 4 files in the browser: inexistent file (404), HTML file with image, PDF file, PNG file
- (If you made the client) How you run the client, show output and saved files
- (If you made directory listing) The directory listing generated page, subdirectory
- (If you browsed your friend's server) Describe your network setup, how you found their IP, screenshots of the contents of their server, screenshots of requests to their server (using your own client if you implemented it)

## Lab 2: Concurrent HTTP server



The questions will be based on:

- <https://web.mit.edu/6.102/www/sp25/classes/14-concurrency/>
- Section 4.3 from [this article](#)
- The definitions from [this glossary](#)

- (Optional) *The Art of Multiprocessor Programming* (you can find a copy in the Drive)

## A bit of theory

There are two distinct schools of thought that use the same words differently. As a consequence, the answer to “does parallel imply concurrent?” depends on the school of thought:

- In the **OS** tradition:
  - Concurrency = tasks overlap in time (even if interleaved)
  - Parallelism = tasks run simultaneously (on multiple processors)
  -  Parallel  $\Rightarrow$  Concurrent
    - All parallel tasks are also concurrent
    - Not all concurrent tasks are parallel
  - common in the OS world
- In the **PLT** (Programming language theory) tradition:
  - Concurrency is a language concept: constructing a program as independent parts
  - Parallelism is a hardware concept: executing computations on multiple processors simultaneously
  -  Parallel  $\neq$  Concurrent
    - They are orthogonal concepts
    - A concurrent program may or may not execute in parallel
    - A parallel computation may or may not have the notion of concurrency in its structure.
- Found in all the linked readings

As you will see, all the linked resources abide by the second school of thought. If you only know about the first school of thought, I will consider your answer wrong.

## The task

In this lab, you will make your HTTP server multithreaded, so that it can handle multiple connections concurrently. You can either create a thread per request, or use a thread pool.

To test it, write a script that makes multiple concurrent requests to your server. Add a delay to the request handler to simulate work ( $\sim 1s$ ), make 10 concurrent requests and measure the amount of time in which they are handled. Do the same with the single-threaded server from the previous lab. Compare the two.

## Counter: 2 points

Add a counter feature. Record the number of requests made to each file and show it in the directory listing. For example:

# Directory listing for /

File / Directory	Hits
<a href="#">.DS_Store</a>	1
<a href="#">computer-networking-a-top-down-approach-8th-edition.pdf</a>	132
<a href="#">computer_networks_5th_toc.pdf</a>	86
<a href="#">computer_networks_6th.pdf</a>	174
<a href="#">Pearson/</a>	59
<a href="#">Theartofmulticore.pdf</a>	201

First, implement it in a naive way and show that there is a race condition (you can rewrite the code and add delays to force interlacing of threads). Then, use a synchronization mechanism (e.g. a lock) and show that the race condition is gone.

## Rate limiting: 2 points

Implement rate limiting by client IP (~5 requests/second) **in a thread-safe way**. Have one friend spam you with requests and another send requests just below the rate limit. Compare the throughput for both (successful requests/second).

## Lab 3

**Work in progress**

## Lab 4: Multiplayer Game

**Work in progress**

You will implement the [MIT 6.102 \(2025\) Memory Scramble lab](#).

A multiplayer game is a great lab for a PR course because it teaches important concepts in a fun way:

- Concurrency
- Implementing a Web API

## Curriculum

Here is the list of topics that will be covered in this course:

1. Concurrency
  - a. Threads
  - b. Locks
2. Network programming
  - a. Sockets, TCP, UDP
  - b. Protocols, HTTP
  - c. Serialization (JSON)
3. Git
4. Docker

## Sources

It's better to make use of material made by professional educators than to come up with your own.

- "Computer Networking: a Top Down Approach" (Pearson, )
  - Has [labs in python and slides](#) for teaching.
  - [Interactive animations](#)
  - Sockets, web servers, protocols, proxies, etc.
  - Doesn't explain concurrency.
  - Lab 1: HTTP server with TCP sockets
    - Additional 1: make it multithreaded
    - Additional 2: write HTTP client
  - Lab 2: UDP client + server, timeouts
  - Lab 3: implement basic SMTP protocol
  - Lab 4: implement Ping using raw sockets (ICMP)
    - Lab 4.2: implement traceroute using ICMP
  - Lab 5: HTTP proxy server
  - Lab 6: video streaming, implement RTSP with sockets
- University Course [MIT 6.102 "Software Construction" \(2025\)](#)
  - Concurrency (high-level):
    - [15: Promises](#)
    - [16: Mutual Exclusion](#)
    - Message passing
  - Low-level concurrency in older course: [MIT 6.005 "Software Construction" \(2016\)](#):
    - Thread safety
    - Locks & synchronization
- The definitive [Glossary of concurrency concepts by sliks](#)
- A terse and elegant overview of concurrency (independence) and concurrency paradigms without observable nondeterminism: [Programming Paradigms for Dummies: What Every Programmer Should Know by Peter Van Roy](#)
- A comprehensive book on the concepts of concurrency: ["The Art of Multiprocessor Programming"](#) (2008) (multiprocessor = multi-core):
  - Theoretical (Mutual Exclusion)
  - Practical (Locks, Monitors)