A3: Computer Networking

Computer Systems 2025-26

Department of Computer Science
University of Copenhagen

David Marchant

Due: Sunday, 23rd of November, 16:00
Version 1 (October 24, 2025)

This is the fourth assignment in the course on Computer Systems 2025 at DIKU and the only one within the topic of Computer Networking. We encourage pair programming, so please form groups of 2 or 3 students. Groups cannot be larger than 3, and we strongly recommend that you do not work alone.

For this assignment you will receive a mark out of 6 points; You must attain at least half of the possible points on this assignment to be admitted to the exam. For details, see the *Course description* in the course page. Resubmission is not possible.

It is important to note that an attempt must be made for the programming task, the report and the theory questions. Not having a submission for each will result in 0 points. Therefore, although you can obtain 50% of the marks just from the report part, you still require a theory and programming part.

The web is more a social creation than a technical one. I designed it for a social effect — to help people work together — and not as a technical toy. The ultimate goal of the Web is to support and improve our weblike existence in the world. We clump into families, associations, and companies. We develop trust across the miles and distrust around the corner.

— Tim Berners-Lee, Weaving the Web (1999)

Overview

This assignment has three parts, namely a theoretical part (Section 1), a programming part (Section 2)., and a report part (Section 4). The theoretical part deals with questions that have been covered by the lectures. The programming part requires you to complete a peer to peer file transfer service using socket programming in C. The report part is a short report about the programming part detailing your design decisions. In this assignment, the programming effort relies on building a peer consisting of concurrent client and server interactions. Much of this assignment is designed to work in conjunction with

code developed during the exercise sessions. This will be highlighted as the appropriate points. More details will follow in the programming part (Section 2).

1 Theoretical Part (25%)

You should also hand-in answers to the following questions. These questions are each quite open-ended, and unless otherwise noted should take only a paragraph or two to answer.

1.1 Reserving Connections

At the start of the networking section of the course we introduced circuit-switching which reserves a connection across the network. We explained that this was inefficient and instead the modern internet uses a connectionless packet-based system instead. Later on we explained how most communication across the network is done via TCP, which is connection oriented. How is it true that our modern internet is both connectionless and connection dependent?

1.2 Reliable Communication

This course has often presented as though all messages are sent either TCP or UDP. This is largely true, but there has been a trend in recent years of creating TCP alternatives, on top of UDP. These are something similar to the protocols in this assignment, e.g. application layer protocols that try to create some semblance of reliable communication manually, without the overhead of TCP. What must these protocols implement, and how would you expect them to achieve reliable communication using the fundamentally unreliable UDP.

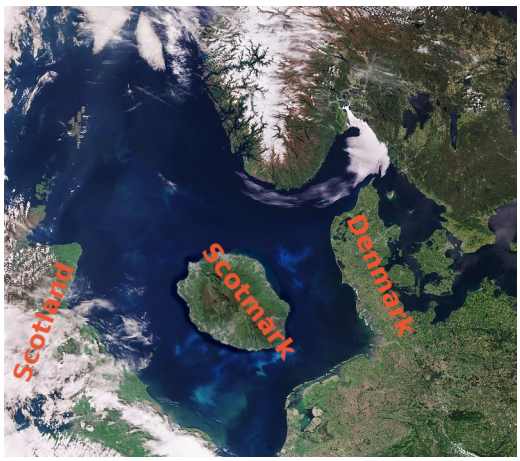
1.3 DNS

Breaking news! The island nation of Scotmark has been noticed for the first time, located in the middle of the North Sea (See satellite image below for proof). The nation is conveniently very culturally and technologically similar to Denmark, and is already running servers hosting websites to attract tourists. Unfortunately the nation of Scotmark has not discovered DNS, and so any network addresses aren't discoverable without users manually typing IP addresses. Assign the country a suitable top-level domain, and describe what further steps should be taken so that the wider internet can access all Scotmarks websites from their human-memorable URLs. You should assume in your answer that numerous large companies, universities, government organisations etc already exist in Scotmark.

1.4 Hashing

Rainbow tables only work if we hash inputs using known hash functions. Rather than bothering with salts, an alternate solution could be to use a hash function uniquely developed for each application. Why is this not advised? You should explain what features a hash function is expected to have, and how this approach might compromise those features, as well as any other additional relevant points.

Figure 1: A satelite view of beautiful Scotmark.



1.5 Deadlock

There are two common computer science problems both known as 'The Santa Claus Problem'. The first deals with routing around locations and is also known as the traveling salesman, but the second is to do with concurrency and is defined in this paper. Pseudo-code for a proposed solution is found below, your task here is to assess if the system will deadlock or not. You should provide diagrams or other notes, showing how you have identified potential deadlocks. You should then reason about if this is a genuine potential deadlock or not. Note that as there are several different components at the same time, depending on how you approach this problem, you *may* find it easier to examine certain combinations of components rather than the whole system. Additionally, note that you do not need to comment on the correctness of the solution, only its potential for deadlock.

```
all_reindeer = []
reindeer_mutex = mutex()
busy_mutex = mutex()
elf_mutex = mutex()

stable_handler(connection)
    reindeer_address = connection.read()
    lock(reindeer_mutex)
    all_reindeer.append(reindeer_address)
    if all_reindeer.length == 9 //If all reindeer in list
        santa_connection = socket(santa_address)
        santa_connection.write("REINDEER READY")
    unlock(reindeer_mutex)

stable(my_address, santa_address)
    listening_socket = socket(my_address)
    while (true)
        connection = listening_socket.accept()
```

```
        thread(stable_handler, (connection))

reindeer(my_address, stable_address)
    listening_socket = socket(my_address)
    stable_connection = socket(stable_address)
    stable_connection.write(my_address)
    while (true)
        connection = listening_socket.accept()
        message = connection.read()
        if message == "START DELIVERIES":
            print("Reindeer is out on deliveries")
        if message == "DONE DELIVERIES":
            print("Reindeer is now on their holiday")
            sleep(random(40,120)) //Sleep between 40 and 120 seconds
            stable_connection.write(my_address)

elf(my_address, santa_address)
    listening_socket = socket(my_address)
    santa_connection = socket(santa_address)
    while (true)
        sleep(random(10,1000)) //Sleep between 10 and 1000 seconds
        lock(elf_mutex)
        santa_connection.write(my_address)
        unlock(elf_mutex)
        connection.read()
        print("Elf is consulting Santa")
        connection.read()
        print("Consultation done")


santa(my_address)
    listening_socket = socket(my_address)
    waiting_elves = []
    while (true)
        connection = listening_socket.accept()
        message = connection.read()
        switch message:
            case "REINDEER READY":
                lock(busy_mutex)
                lock(reindeer_mutex)
                for reindeer_address in all_reindeer:
                    reindeer_connection = socket(reindeer_address)
                    reindeer_connection.write("START DELIVERIES")
                sleep(40)
                for reindeer_address in all_reindeer:
                    reindeer_connection = socket(reindeer_address)
                    reindeer_connection.write("DONE DELIVERIES")
                all_reindeer = []
                unlock(busy_mutex)
                unlock(reindeer_mutex)
            default:
                waiting_elves.append(message)
```

```
                if (waiting_elves.length == 3): //Only wait for 3 elves
                    lock(busy_mutex)
                    for elf_address in waiting_elves:
                        elf_connection = socket(elf_address)
                        elf_connection.write(1)
                    lock(elf_mutex)
                    sleep(20)
                    unlock(elf_mutex)
                    for elf_address in waiting_elves:
                        elf_connection = socket(elf_address)
                        elf_connection.write(1)
                    waiting_elves = []
                    unlock(busy_mutex)

main()
    santa_address = 10000
    stable_address = 20000
    reindeer_addresses = 30000
    elf_addresses = 40000
    Thread(santa, (santa_address)) //Start santa thread
    Thread(stable, (stable_address, santa_address)) //Start stable thread
    for i in range(9): // Start 9 reindeer threads
        Thread(reindeer, (reindeer_addresses+i, stable_address))
    for i in range(12): // Start 12 elf threads
        Thread(elf, (reindeer_addresses+i, santa_address))
```

2 Programming Part (25 %)

For the programming part of this assignment, you will implement a peer in a peer-to-peer file-sharing service. This is a substantially larger tasks than previous assignments and so is designed to go along with the corresponding exercise classes. These links will be highlighted in the relevant sections.

2.1 Design and overview

The file-sharing service consists of many identical peer processes. When a peer starts it must be given its own address, which it can then tell to others so they can connect back to it. Once started, each peer will ask for the address of another peer which it can attempt to connect to, and so join the network. The first peer to join will not attempt to connect to anyone, as there is no one yet to connect to.

This design uses a custom protocol outlined below. It has *some* security features built in, which you should use appropriately. By this it is meant that you should be able to explain how your implementation keeps to these specifications. In this protocol each peer can perform either client or server interactions, and must be capable of performing each concurrently. Note that this may also mean that different server interactions may be performed concurrently within the same peer.

The first peer to join a network cannot perform client interactions, as there would be no peers to interact with. Therefore, it will only perform server interactions. We shall refer to this as the *initial peer*, though do note that it is programmatically identical to any other peer. If a new peer, referred to here as the *joining peer* were to attempt to join the network it would first contact the *initial peer* to make itself known to the network. The *initial peer* will then update its record of all peers on the network and reply to the *joining peer* with a complete list of all peers on the network. The *initial peer* will then inform any other peers on the network that the *joining peer* has joined, so that all peers will have an up-to-date record of all peers on the network. If a third peer or fourth peer joined they could do so in the same way, using any peer already on the network as their initial peer to contact.

When it is time for a peer to retrieve a file, it will randomly select a peer from the network and send it a request for the file. The responding peer will then respond appropriately, either informing that it does not have the file, or if it does, it will send it to the requesting peer. All messages are limited in how many bytes can be sent in a single message. If any message is too long it will be broken into several messages which the recipient will have to reassemble.

For this assignment you are going to implement a peer that registers with a network, retrieves files, and responds appropriately to those same requests. You can find a complete description of the protocol in Section 3.

2.2 API and Functionality

2.2.1 Key Implementation Tasks

This is a large and complex project, that is expected to fill three weeks of work. In order to help guide you through this, 5 key progress tasks have been identified. Each should be completed according to the protocol described in Section 3. Each of these tasks can themselves be quite large, and so accompanying exercises have been linked that will partially guide you through completing them. These tasks are:

2.1. A peer should be able to register on the network by assembling a registration message and parse the received response. Guidance is available in the exercises for 27/10/25.

2.2. Correspondingly, the peer should also be capable of receiving registration requests from others and generating appropriate responses to them Guidance is available in the exercises for 29/10/25.

2.3. Upon receiving a request to join the network from a new peer, messages informing others on the network of the new peer should also be generated and sent. Guidance is available in the exercises for 03/11/25.

2.4. A peer should be capable of generating a new request for a file, and should be capable of responding to the same calls from other peers. Guidance is available in the exercises for 05/11/25.

2.5. Finally, the peer should be capable of breaking up large message over several messages and reordering them correctly if necessary. This will enable the sending and receiving of files larger than the arbitrary message size limit. Guidance is available in the exercises for 17/11/25.

Note that the dates for the available guidance exercises should not be taken as a marker of how long I expect each tasks to be completed in. Some tasks will take considerably longer than others so have a look through them all and plan your time accordingly. By the time this assignment has been handed out, you will already be familiar with all necessary programming constructs, so there is no reason to wait for later exercises before starting a task.

2.2.2 Test environment

One of the difficulties with developing a system that relies on network communication is that besides a network, you need to have two or more peers running, and potentially debug both simultaneously.

To assist in developing and testing your code, we have provided an implementation of the peer, but written in Python. You can use this to get started with a setup that runs fully contained on your own machine. By running a local Python peer and your own C peer it is possible to debug and monitor all pieces of the communication, which is often not possible with an existing system.

Note that there are two directories within the Python directory, *first_peer* and *second_peer*. This is as you should be testing two sides of interaction with your C peer. E.g. test that your C peer can register with the *first_peer* and retrieve files from it. Also test from the *second_peer* that it can request from your C Peer and have files sent to it when requested. You should also test with all 3 at the same time so as to test your concurrency setup as you CANNOT assume that all client interactions will complete before all server interactions start or vice versa. Adding a sleep into the server interactions will slow down responses, allowing you to check that multiple clients can connect and be responded to at the same time. Note that both Python peers are identical (technically they should be literally the same file just linked). You may wish to add further duplicates depending on what you want to test.

You can of course also peek into or alter the Python and get inspiration for implementing you own peer, but beware that what is a sensible design choice in Python *may* not be a good choice for C. You are also reminded that although you can alter the Python as much as you want to provide additional debugging or the like, they are currently correct implementations of the defined protocols. Be careful in making changes that you do not alter the implementation details, as this may lead your C implementation astray.

To start an appropriate test environment you can run a peer using the command below, run from the *python/first_peer* directory:

```
python3 ./peer.py 127.0.0.1 12345
```

This will start a peer at 127.0.0.1:12345. This will be capable of serving any files within the *python/first_peer* directory. You can now start another peer to connect to this one. You could use either a second Python peer within *python/second_peer*, or the C.

To make and deploy the C peer run the following within the *src* directory:

```
make
./peer 127.0.0.1 23456
```

This will compile your client, though as not yet implemented it will not do anything significant.

Do note that both the Python and C peers as handed out act as two concurrent flows of control, one for server interactions and one for client. For ease of development you may wish to separate the two entirely and have peers acting purely as servers and some purely as clients. This is fine for development, but the final implementation will be expected to act as both at any one time.

2.3 Run-down of handed-out code and what is missing

- *src/peer.c* contains the peer, and is where it is expected all (or at least most) of your coding will take place. If you alter any other files, make sure to highlight this in your report. This code contains some basic setup for your implementation, namely collecting the user input and starting a client and server thread. These should illustrate where you might begin, but feel free to make whatever alterations to the structure you wish.

Your edits WILL NOT be limited to these functions, and you will NEED to make additional functions if your code is to be considered readable. Your completed system should write any retrieved files to the *src* directory.

- *src/peer.h* contains a number of structs you *may* find useful in building your client. You are not required to use them as is, or at all if you would rather solve the problem some other way.
- *src/sha256.c* and *src/sha256.h* contain an open-source stand-alone implementation of the SHA256 algorithm, as used throughout the network. You should not need to alter this file in any way.
- *src/common.c* and *src/common.h* contains some helper functions to assist in parsing some expected data, and for hashing some inputs file. Before you start your own coding have a look through the provided functions to make sure you don't waste time reimplementing functionality that has already been given to you. You should not need to alter this file in any way.
- *src/compsys_helpers.c* and *src/compsys_helpers.h* contains definitions for robust read and write operations. You should not need to alter this file in any way.
- *src/endian.h* contains definitions for converting between big and small endian numbers. This is only useful on a mac, and should not be necessary in the majority of solutions so do not be concerned if you do not use any of these functions. You should not need to alter this file in any way.
- *python/first_peer/peer.py* and *python/second_peer/peer.py* contains the peer, written in Python. This can be run on Python 3.7 or newer. You should not need to alter this file, though may find adding more debug print statements helpful.
- *python/first_peer/tiny.txt* and *python/first_peer/hamlet.txt* are example data files. Testing using additional files would be advantageous in a report.
- `README` contains the various commands you will need to get this project up and running.

2.4 Testing

For this assignment, it is *not* required of you to write formal, automated tests, but you *should* test your implementation to such a degree that you can justifiably convince yourself (and thus the reader of your report) that each API functionality implemented works, and are able to document those which do not.

Simply running the program, emulating regular user behaviour and making sure to verify the result file should suffice, but remember to note your results. When testing parallel systems that may act non-deterministicly any amount of user testing is of limited utility. However, you should attempt to

stress-test your system say by running many peers at once, or by maximising chances of race conditions.

One final resource that has been provided to assist you is that an instance of peer.py that has been remotely hosted, and which you can connect to. This is functionally identical to those contained in the handout, with the obvious difference that any communications with it will be properly over a network.

The server is designed to be robust, so should be resilient to any malformed messages you send, but as it is only a single small resource be mindful of swamping it with requests and only use them once you are confident in your system. Having said that, occasionally it will be broken by some errant message I have not predicted. If you think you've broken it, or it's just not responding then please let me (David) know. You can connect to the peer at the following address and port:

```
Test Server: 130.225.104.168:5555
```

Note that depending on whatever firewalls or other network configuration options you have, you may not be able to reach the server from home, but that you should be able to from within the university. Additionally, note that this version of the Peer will slightly differ from the presented Python peer, in that it will automatically remove peers from its network it has not heard from for 5 mins. This will prevent old test trackers from clogging up the network and is not required functionality for you to emulate.

3 Protocol description

This section defines the implementation details of the components used in the A3 peer-to-peer file-sharing network.

3.1 File structures of the peer

All peers will be able to serve/write files relative to their own locations. In other words, any files in the same directory as a peer should be servable and will be written to the directory of a peer.

3.2 Security considerations

In order to protect user-remembered passwords, they are always salted and then hashed. This is referred to as a signature within this protocol. The signature is treated effectively as a password by the system at large. The user-remembered password should NEVER be transmitted across the network. The salts used here would normally be generated randomly, but to make the implementation easier to debug they are currently implemented as hard coded salts.

When a peer is told of others in the network it will save the contact details of the other peers as their address(including both the IP and port) and signature

derived from their user-remembered password. The signature that peer provides to the system will not be directly saved, but is salted and hashed again to form a final, saved signature. The first peer to handle registering a new peer will randomly generate a salt, and so the salt used will be stored alongside so that the saved signature can be re-derived to check subsequent signatures.

3.3 Format of the peer requests

All messages from the peer must contain the same header as follows:

```
16 bytes - The IP address the sending peer is listening on,
           as UTF-8 encoded bytes
4 bytes - The port the sending peer is listening on,
          unsigned integer in network byte-order
32 bytes - Signature, a hash of the salted peer password,
           as UTF-8 encoded bytes
4 bytes - The command code desribing the nature of this
          message, unsigned integer in network byte-order
4 bytes - The length of the request body, unsigned integer
          in network byte-order
```

The command codes specified above can only be one of 3 values. These, along with the appropriate request bodies for them are:

```
1 - Register, when a peer is attempting to make first
    contact with a network. No body is provided with this
    command
2 - Retreive, when a peer is requesting a data file to be
    transfered to it from another peer. The request body
    must be the filename/filepath of the requested file
3 - Inform, when a peer is informing another peer of a
    third peer that has just joined the network, The
    request body must fit the 68 bytes described below:
 16 bytes - The IP address of the newly joined peer,
            as UTF-8 encoded bytes
  4 bytes - The port of the newly joined peer, unsigned
            integer in network byte-order
    32 bytes - The peer-saved signature of the newly joined
               peer, that being a salted and hashed
               signature, as UTF-8 encoded bytes
    16 bytes - The salt used in the above signature,
               as UTF-8 encoded bytes
```

In the case of the Register and Retrieve (1 & 2) commands, or if the command could not be determined, a reply will ALWAYS be expected from the other peer. ONLY in the case of the Inform (3) command do we not expect a reply.

3.4 Format of the peer responses

All replies from a peer must contain the same header as follows:

```
4 bytes - Length of response body,
          unsigned integer in network byte-order
4 bytes - Status Code of the response,
          unsigned integer in network byte-order
4 bytes - Block number, a zero-based count of which block in
          a potential series of replies this is,
          unsigned integer in network byte-order
4 bytes - Block count, the total number of blocks to be sent,
          unsigned integer in network byte-order
32 bytes - Block hash, a hash of the response data in this
           message only, as UTF-8 encoded bytes
32 bytes - Total hash, a hash of the total data to be sent
           across all blocks, as UTF-8 encoded bytes
```

In addition to the header each response will include an additional payload of the length given in the header. Messages are limited to a total length of 8196 bytes, including both the header and the body. Note that in the case of a small enough response to only require a single reply, then the block hash and total hash will be identical. If the response is to a request for a data file, and it could be retrieved with no errors then the payload will be either all or part of said file, depending on the size of the file. If the response is to message format 1 (e.g. a request to register), the response will be a list of all known peers on the network. Each entry in the list will be formatted as:

```
16 bytes - The IP address of a network peer,
           as UTF-8 encoded bytes
4 bytes - The port of a network peer,
          unsigned integer in network byte-order
32 bytes - Signature, the peer-saved signature, that being a
           salted and hashed signature, as UTF-8 encoded bytes
16 bytes - Salt, the salt used in the above signature,
           as UTF-8 encoded bytes
```

The list itself will be some multiple of 68 bytes long, with the first 68 bytes being one peer, the second being another and so on. Note that it is not defined behaviour if this list includes the joining peer, and it is up to you to ensure your network list does not contain duplicate peers. In all other cases the response will be a feedback message explaining any errors or results of other queries.

The following status codes may be provided in response by the peer:

```
1 - OK (i.e. no problems encountered)
2 - Peer already exists (i.e. could not register a peer as
    they are already registerd with the network)
3 - Peer is missing (i.e. a non-register request was
    encountered from a peer who has not yet registered)
4 - Password mismatch (i.e. the provided signature differs
```

```
    from the one provided during registration)
5 - Bad Request (i.e. the request is coherent but cannot be
    servered as the file doesn't exist or is busy)
6 - Other (i.e. any error not covered by the other status
    codes)
7 - Malformed (i.e. the request is malformed and could not be
    processed)
```

3.5 Requests/Responses and persistent connections

Note that there are no persistent connections between peers. Each time a peer sends a request, the serving peer will respond with one or more reply messages, but it will then close the connection. To request multiple files, a peer will need to open a separate connection each time.

4 Report Part (50 %)

4.1 Report and approximate weight

Your assignment submission must include a report details your implementation. Although this is worth 50% of the marks, you may find it very hard to complete without any implementation. Similarly, do not assume it will take 50% of your time, I expect most of it will be spent on the implementation. You should also find that you do not need to complete your implementation before you can start writing most of this.

Please include the following points in your report:

- Discuss the provided protocol and how you were able to ensure a thread-safe implementation. For instance, what data needed to be shared between the client and server side of the application, and how did you avoid race conditions? Have you done so in a way that also avoids deadlocks? What circumstances would stress your implementation either by creating errors, or by degrading performance? (Approx. weight: 10%)
- Document technical implementation you made for the peer - cover each of the five implementation tasks outlined in Section 2.2.1, as well as any additional changes you made. Each change made should be briefly justified. For each of the five tasks you should describe what design principles you have stuck to and what the effects of doing so have been. For instance, you could discuss how you have ensured good scalability, correctness, throughput or any other metric you feel is important. (Approx. weight: 20%)
- Discuss how your design was tested. What data did you use on what machines? Did you automate your testing in any way? Remember that you are not expected to have built a perfect solution that can manage any and all input, but you are always meant to be able to recognise what will break your solution. Testing on the provided two files (tiny.txt and hamlet.txt) is not sufficient, you should also test using data that does not work, or with requests that are incorrect. You should also test with more than 2 peers running at once. You should also describe what input/data your implementation is capable or processing and what it is not.(Approx. weight: 10%)
- Discuss any shortcomings of your implementation, and how these might be fixed. It is not necessarily expected of you to build a fully functional peer, but it *is* expected of you to reflect on the project. Even if you implement the protocol exactly as described throughout this document there will still be problems to identify in terms of usability, concurrency or functionality. (Approx. weight: 10%)

As always, remember that it is also important to document half-finished work. Remember to provide your solutions to the theoretical questions in the report pdf.

Submission

You should hand in a ZIP archive containing a `src` directory containing all *relevant* files (no ZIP bomb, no compiled objects, no auxiliary editor files, etc.).

Any Python code should also be included in your submission as it may form part of how you tested your C code (hint). As this course is not a Python course, you will not be marked according to the quality of your Python code.

To `make` this a breeze, we have configured the `src/Makefile` such that you can simply execute the following shell command within the `src` directory to get a `zip`:

```
$ make zip
```

Note that depending on any test files, or how you implemented your solutions you may wish to include other files not zipped by the above command. You are allowed to do so, but are encouraged to be sensible in what you include (E.g. please no 10GB test files).

Alongside a `src.zip` submit a `report.pdf`. For submission, sign up for a group at:
`https://absalon.ku.dk/courses/70194/groups#tab-23701`
Hand-in as a group as in other courses and remember to include your AI declaration