# A3: Computer Networking (I)

## Computer Systems 2022
### Department of Computer Science
### University of Copenhagen

David Marchant

**Due:** Sunday, 20th of November, 16:00
**Version 1** (October 30, 2022)

---

This is the fourth assignment in the course on Computer Systems 2022 at DIKU and the first on the topic of Computer Networks. 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 4 points; You must attain at least half of the possible points to be admitted to the exam. For details, see the *Course description* in the course page. This assignment belongs to the CN category together with A4. Resubmission is not possible.

It is important to note that only solving the theoretical part will result in 0 points. Thus, the theoretical part can improve you score, but you *have* to give a shot at the programming task.

---

*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 two parts, namely a theoretical part (Section 1) and a programming part (Section 2). The theoretical part deals with questions that have been covered by the lectures. The programming part requires you to complete a file transfer service using socket programming in C. In this assignment, the programming effort relies solely on building the client portion of the architecture. More details will follow in the programming part (Section 2).

# 1    Theoretical Part (25%)

Each section contains a number of questions that should be answered **briefly**
and **precisely**. Most of the questions can be answered within 2 sentences or
less. Annotations have been added to questions that demand longer answers,
or figures with a proposed answer format. Miscalculations are more likely to
be accepted, if you account for your calculations in your answers.

## 1.1    Store and Forward

The answers to the questions in this section should not make any assumptions
about specific protocols or details pertaining to the different layers. You can
answer these questions after having read Chapter 1 of the K&R book.

### 1.1.1    Processing and delay

Explain, within three or four sentences, the reasons for delay in packet switched
networks, besides physical constraints such as the propagation speed of differ-
ent transmission media.

### 1.1.2    Transmission speed

Consider the setup below in Figure 1. A DIKU student is using a laptop at
home, browsing the `diku.dk` website. The upstream connection speed is 2
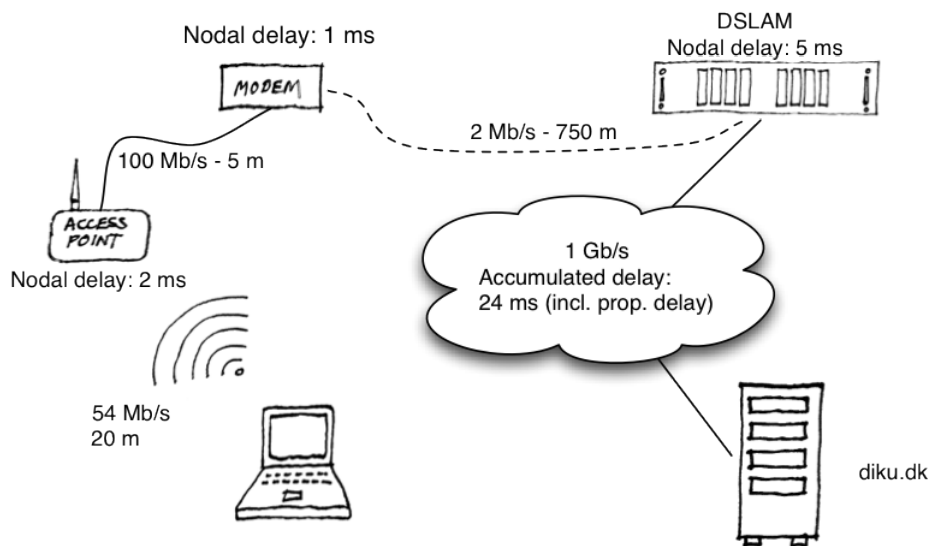Mb/s from the DSL modem at home to the DSLAM[1].



**Figure 1:** A typical DSL setup

---

[1]Digital Subscriber Line Access Multiplexer, used by Internet Service Providers to provide DSL
connectivity over phone lines.

**Part 1**　Given the information in Figure 1, calculate the *round trip time* (RTT). For calculating propagation delay, assume that the propagation speed in all links visible is $2.4 * 10^8$ m/s and the queuing delay is contained in the noted node delays. You may leave out the propagation delay, but explain why, if you do.

**Part 2**　Assume 640 KB of data is sent to the `diku.dk` webserver, including any overhead. Assume that the server acknowledges the upload when all bytes have been transfered with a single packet. Calculate the total transmission time, given the RTT calculated above.

## 1.2　HTTP

### 1.2.1　HTTP semantics

HTTP employs a message format divided into header and body sections. The initial header of a request consists of a method field, a resource identifier field and a protocol version field. Likewise, the initial header of a response consists of a protocol version field, a status code and an optional message.

**Part 1:**　What is the purpose of the method field in requests? How do `POST` and `GET` requests differ in practice?

**Part 2:**　An additional (and mandatory) header field is the `Host` header. Why is this header necessary?

### 1.2.2　HTTP headers and fingerprinting

**Part 1:**　One of the additional header fields is `Set-cookie` (for responses) and `cookie`. What is the reason for these header fields and to what degree may they be used as a unique identifier?

**Part 2:**　The `ETag` response header works in conjunction with the `If-None-Match` and `If-Match` request headers to prevent unnecessary page fetches, if enabled. How can ETags work as cookies[2]?

## 1.3　Domain Name System

### 1.3.1　DNS provisions

Three of the most important goals of DNS[3] are to ensure fault tolerance, scalability and efficiency. Explain how these insurances can (and are) met in practice. *(Answer with 2-4 sentences.)*

---

[2]You may want to consult section 13.3.2-3 and section 14.19 `http://www.ietf.org/rfc/rfc2616.txt`.

[3]As specified in RFC 1034 and RFC 1035, superseding RFC 882 and RFC 883.

### 1.3.2   DNS lookup and format

**Part 1:**   Explain the advantages of the `CNAME` type records.  Explain how DNS may provide simple load balancing among servers. *(Answer with 2-4 sentences.)*

**Part 2:**   Many DNS servers, especially *root* and *top level domain* (TLD) servers, respond with 'iterative' replies to recursive requests.  Explain the differences between iterative and recursive lookups and when and why recursive lookups are justified. *(Answer with 4-8 sentences)*

# 2 Programming Part (75 %)

For the programming part of this assignment, you will implement a client in a networked file-sharing service. Many of the programming tasks set in the exercises relate to the sub-tasks within this assignment, so if you are stuck at any point consider revisiting them.

## 2.1 Design and overview

The file-sharing service consists of server and client processes. For this assignment we can assume only one of each, with a single server providing files, and a single peer retrieving them.

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 client should start by registering with the server. This is done by simply telling the server the client is here, which is not a secure methodology in the real world but will suffice for this assignment. If a client attempts any further communication with the server without registering the user, then the server will refuse to provide any requested files. Throughout this protocol we will make repeated use of hashes. These are used for two broad purposes, to obfuscate passwords and login details, and to act as a checksum on our transported data.

The server is limited in how many bytes it can send at once. If the client requests a file larger than this limit then it will therefore be sent over several messages which the client will have to reassemble.

For this assignment you are going to implement a client that registers a new user, and retrieves files and stores them locally.

## 2.2 API and Functionality

### 2.2.1 Key Implementation Tasks

The client you will be implementing needs to perform the four basic tasks listed below. Each should be completed according to the protocol described in Section 3. The client should:

2.1. Create a user signature. This is obtained by salting the user-remembered password and hashing it. This will be used by both the registration and file request messages so the result should either be saved somehow, or be computed in a deterministic manner.

2.2. Register a new user. Both the username and the signature should be sent to the server to register a new user, with the client reporting back the response.

2.3. Retrieve a small file. The user should request a file small enough to fit into a single message. For this the example file tiny.txt has been provided,

though you are free to test on others. This should be retrieved by the client and written into its own file storage. The client should validate the recieved file to check that it has been recieved correctly.

2.4. Retrieve a large file. The user should request a file large enough to require several message to completely send. For this the example file hamlet.txt has been provided, though you are free to test on others. This should be retrieved by the client and written into its own file storage. The client should validate the recieved file to check that it has been recieved correctly.

### 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 both the client and server running, and potentially debug both simultaneously.

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

You can of course also peek into or alter the Python client and get inspiration for implementing you own client, 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 do not alter the implemenation details, as this may lead your C implementation astray.

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

```
python3 server.py ../../config.yaml
```

Assuming you have not altered `config.yaml`, this will start a server at 0.0.0.0:12345. This will be capable of serving any files within the *python/server* directory. You can then start an ideal client by running another command prompt from the *python/client* directory:

```
python3 client.py ../../config.yaml
```

This will start a client capable of registering a new user and requesting the two files, and will host at 0.0.0.0:23456. This will write any retrieved files to the *python/client* directory.

The C client can be started by calling the following from within the *src* directory:

```
make
./networking ../config.yaml
```

By default this will compile your client, though as not yet implemented it will not do anything.

Do note that both the Python and C clients as handed out act as simple linear scripts, registering a user and then retrieving the two required files. This structure has been picked for ease of development and understandability only, and is not intended as demonstration of good practice. If you would rather alter this structure so as to allow for a client to dynamically request whatever files they wish, then do feel free to do so, though it is not aim or requirement of this assignment.

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

- *src/networking.c* contains the client, and is where it is expected all of your coding will take place. If you alter any other files, make sure to highlight this in your report. This code contains three guide functions, `get_signature`, `register_user` and `get_file` to illustrate where you might begin, but feel free to make whatever alterations to the structure you wish. Your completed system should write any retrieved files to the *src* directory.

- *src/networking.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 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 the configuration file. You should not need to alter this file in any way.

- *src/csapp.c* and *src/csapp.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/server/server.py* contains the server, written in Python. This can be run on Python 3.7 or newer, and requires a config file written in the YAML format. You should not need to alter this file, though may find adding more debug print statements helpful.

- *python/client/client.py* contains the client, written in Python. This can be run on Python 3.7 or newer, and requires a config file written in the YAML format. You should not need to alter this file, though may find adding more debug print statements helpful.

- *python/shared.py* contains some functions shared between the server and client. You should not need to alter this file, though may find adding more debug print statements helpful.

- *config.yaml* contains the definitions for where the server and client will be hosted.

- *python/server/tiny.txt* and *python/server/hamlet.txt* are example data file which if you can transfer to the client correctly are enough to say you have completed this part of the assignment. Testing using addition files would be advantagous 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. Just testing using the three function calls (e.g. register a correct user, retrieve a small file, retrieve a large file) is not enough to say the system works. Other tests such as retrieving without first registering, or requesting a file that does not exist should also be run at a minimum.

One final resource that has been provided to assist you is that an instance of server.py 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 propperly 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. You can connect to the tracker at the following address and port:

        Test Server: 130.225.104.138:5555

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.

## 2.5   Recommended implementation progress

As mentioned in the previous section, `src/network.c` contains two unimplemented functions, the implementation of which should yield a functioning client. Do note that you are not limited to completing these functions and you are free to make whatever changes, additions or removals you wish at any point in the provided C code, as long as you successfully implement the desrcibed protocols. It is expected you will stick to modifying network.c, and should specifically highlight in your report if you deviate from this.

In this section, we give a short recap of your implementation todos; this can serve as a checklist for your project, and *we recommend* that you work on them in the order presented here.

2.1. Assemble the user signature from a password and salt

2.2. Register a user with the server and print any feedback to the registration request

2.3. Request a small file from the server and either create a local copy of the file, or print any feedback if an error is encountered

2.4. Request a large file from the server and either create a local copy of the file, or print any feedback if an error is encountered

2.5. Manually test your implementation, documenting bugs found and how you fix them (if you are able to).

2.6. Meanwhile, do not neglect the theoretical questions. They may be relevant to your understanding of the implementation task.

## 2.6   Report and approximate weight

The following approximate weight sums to 75% and includes the implementation when relevant.

Please include the following points in your report:

- Discuss the provided protocol and how you were able to use it. For instance, what security assurances were you able to make using it? What gaps exist in this security setup and what changes, improvements or expansions should be made to the protocol to help address them? (Approx. weight: 15%)

- Document technical implementation you made for the client - cover in short each of the three tasks (register, small file, large file), as well as any additional changes you made. Each change made should be breifly justified. You should also describe what input/data your implementation is capable or processing and what it is not. (Approx. weight: 30%)

- 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 recongise 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. (Approx. weight: 15%)

- Discuss any shortcomings of your implementation, and how these might be fixed. It is not necessarily expected of you to build a fully functional client, but it *is* expected of you to reflect on the project. Even if you implement the protocol exactly as described through this document there will

still be problems to identify in terms of usability, security or functionality.
(Approx. weight: 15%)

*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.*

# 3    Protocol description

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

## 3.1    File structures of the server and client

Both the server and client will be able to serve/write files relative to their own
locations. In other words, any files in the same directory as the client or server
should be servable.

## 3.2    Security considerations

In order to protect user-remembered passwords, they are always salted and
then hashed. This is refered 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 server will register a user by username and signature. The server will
save this signature by salting it again, and hashing. This will then be saved
along with the salt used by the server.

## 3.3    Format of the client requests

All messsages from the client to the server must contain the same header as
follows:

```
16 bytes - Username, as UTF-8 encoded bytes
32 bytes - Signature, a hash of the salted user password,
           as UTF-8 encoded bytes
 4 bytes - Length of request data, excluding this header,
           unsigned integer in network byte-order
```

If the message is intended as a registration of a new user then no additional
data is provided, and so the length of the request data will be zero.

If the message is intended as a request for a file, then an additional field
of the length given in the header will also be present. This is a path to the
requested file to be retrieved.

Regardless of the message type, a response will ALWAYS be expected from
the server.

### 3.4   Format of the server responses

All replies from the server to the client must contain the same header as follows:

```
4 bytes - Length of response data, excluding this header,
          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. 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 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. In all other cases the response will be a feedback message explaining any errors or results of other querries.
  The following error codes may be provied by the server:

```
1 - OK (i.e. no problems encountered)
2 - User already exists (i.e. could not register a user as
    they are already registerd)
3 - User missing (i.e. could not service the request as the
    user has not yet registered)
4 - Invalid Login (i.e. the provided signature does not match
    the registerd user)
5 - Bad Request (i.e. the request is coherent but cannot be
    servered as the file doesn't exist)
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)
```

# Submission

The submission should contain a file `src.zip` that contains the `src` directory of the handout; this should include any and all files necessary to run your code (including `csapp.c, csapp.h,` your `Makefile,` and any new files or test programs that you may have written).

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.

Alongside the `src.zip` containing your code, submit a `report.pdf`, and a `group.txt`. `group.txt` must list the KU ids of your group members, one per line, and do so using *only* characters from the following set:

$$\{\texttt{0x0A}\} \cup \{\texttt{0x30,0x31,...,0x39}\} \cup \{\texttt{0x61,0x62,...,0x7A}\}$$

**Please make sure your submission does not contain unnecessary files, including (but not limited to) compiled object files, binaries, or auxiliary files produced by editors or operating systems.**