# A3: Computer Networking (I)

Computer Systems 2021
Department of Computer Science
University of Copenhagen

Kenneth Skovhede and David Gray Marchant

**Due:** Sunday, 21 of November, 16:00
**Version 1** (November 1, 2021)

---

This is the fourth assignment in the course on Computer Systems 2021 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 fill in the blanks of a file transfer service using socket programming in C. The implementation task of the complete P2P network is spread over this assignment and the next one (A4). 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 different 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].

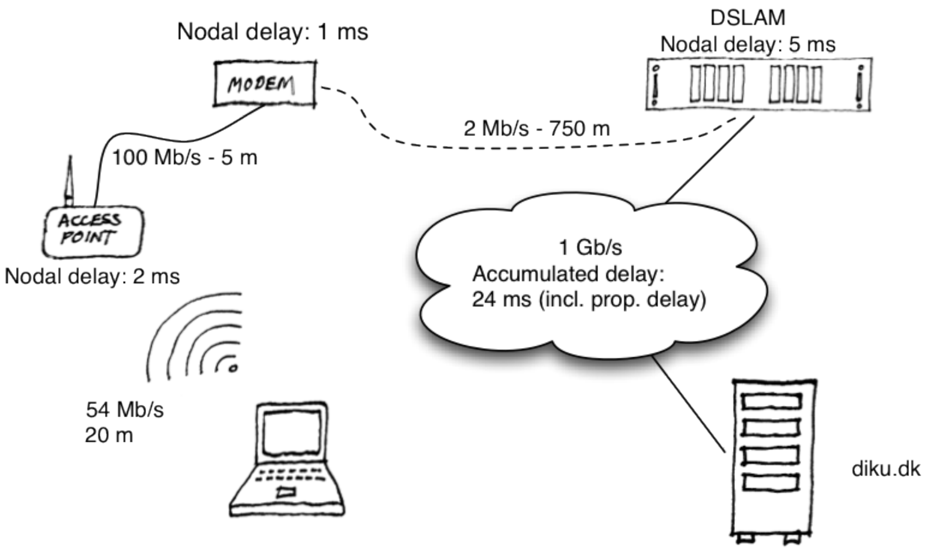

**Figure 1:** A typical DSL setup

[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 peer in a distributed file-sharing service.

### 2.1 Design and overview

The file-sharing service comprises a metadata-file, a tracker, and the peers. Each peer in the real world would act as both a client and a server, but for this first assignment we will only implement the client part, meaning that the *"peer"* will only be able to download the file, not share it with others. In the next assignment, we will add the server capability and form a full peer.

The design of the P2P network is a simpler version of the popular BitTorrent protocol and is named the Cascade protocol, which hints at the relation to torrents.

Like in the BitTorrent protocol, each file that is served on the network is divided into blocks of a fixed size. The client can then choose any of the peers as the source for downloading a particular block. To ensure that the block is correct and not corrupted by the network or a malicious peer, each file is described by a list of hashes[4] in a metadata file, dubbed a *.cascade* file, which can then be used to verify the individual blocks.

Also like the BitTorrent network, the Cascade network relies on a tracker service for discovering peers. A client can retrieve a list of active clients from the tracker as well as register as a peer.

For this assignment you are going to implement a client-only "peer" that parses a `.cascade` file, obtains a list of peers, downloads the blocks required to reconstruct the file, and store the file locally.

### 2.2 API and Functionality

#### 2.2.1 Client-only "peer"

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

2.1. Parse the `.cascade` file - After parsing the file, you will have a structure that contains metadata about the file to be downloaded, including the file size, the size of each block, the number of blocks, and the hash of each block

2.2. Obtain a list of peers - Using the hash of a `.cascade` you can ask the tracker for a list of peers that serve that particular file.

2.3. Download blocks from peers - Download each of the blocks required to restore the file. After obtaining a block, verify that the hash of the block matches.

[4]Hashes will not be covered in depth until later in the course. For now it should be sufficent to understand that a hash is a fixed length numeric representation of a much larger piece of data. For more details consider skimming Section 8.3 of the K&R book.

#### 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 tracker and the full peer, 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 tracker and a local peer 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 peer 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 you do not alter the implemenation details, as this may lead your C implementation astray.

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

- `src/cascade.c` contains the client. At the time of hand-out, the client program supports basic input parsing. The code contains a number `TODO`'s, and it is your job to finish the implementation by filling out the code missing. Each `TODO` is accompanied by one or more hints to get you started. You are of course welcome to expand, alter or otherwise improve parts not marked by `TODO`'s. If you do so, take care that the expeected functionality is still maintained.

- `src/cascade.h` contains a number of useful data structures, defined to contain the various data items needed by the peer.

- `python/tracker/tracker.py` contains the tracker, written in Python. The tracker can run with Python 3.7 or newer, and relies on a config file in JSON format.

- `python/tracker/config.json` contains the configuration details for the tracker. Of particualr note here is the 'allowedhashes' attribute. This lists the hashes of the cascade files the tracker will be capable of tracking. If you generate your own cascade files, make sure to add them here. The provided config file already lists the hashes for the five shakespeare.txt cacscade files.

- `python/peer/peer.py` contains a peer implementation, written in Python. The peer can run with Python 3.7 or newer, and relies on commandline inputs. Next to this peer a number of pregenerated cascade files have been provided for ease of testing.

- `python/filegen/filegen.py` contains the a small program for generating `.cascade` files. To test the system with varying sizes, the tool will generate `.cascade` files for different block sizes.

- `src/sha256.c` and `src/sha256.h` contain an open-source stand-alone implementation of the SHA256 algorithm, as used throughout the network.
- As usual, we provide `csapp.c` and a `Makefile` within the `c` directory.

## 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.

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

The tracker and the peer are designed to be robust, so should be resilient to any malformed messages you send, but as they are each only a single small resource be mindful of swamping them 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 Tracker: 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 tracker from home, but that you should be able to from within the university.

## 2.5 Recommended implementation progress

As mentioned in the previous section, `cascade.c` contains a number of `TODO`'s, the satisfaction of which should yield a functioning peer client. Do note that you are not limited to completing the `TODO`'s 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.

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

2.1. Parse the `.cascade` file

2.2. Check for existing blocks in the local file

2.3. Create a list of missing blocks

2.4. Compute the SHA256 hash of the `.cascade` file

2.5. Request a list of peers serving the hash of the `.cascade` file

2.6. Download the missing blocks from the peers

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

2.8. Meanwhile, do not neglect the theoretical questions :D 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 service guarantees were you able to provide using it? If you feel changes should be made to the protocol, what would they be and why? (Approx. weight: 15 %)
- Document technical implementation you made for the peer - cover in short each of the `TODO`'s for `cascade.c` as well as any additional changes you made. Each change made should be breifly justified. You should also describe what inpur/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. (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 peer, but it *is* expected of you to reflect on the project. (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 `cascade` P2P file-sharing network.

## 3.1 Format of `cascade` files

Each `cascade` files contains information about a single file. The layout of the file header is as follows:

```
8 bytes - "CASCADE1", a string literal
8 bytes - Reserved, SHOULD be zero
8 bytes - Total file length in bytes, unsigned integer in
          network byte-order
8 bytes - Block size in bytes, unsigned integer in network
          byte-order
32 bytes - Complete file SHA256 hash
```

Following the header, the file contains a sequence of 32 bytes SHA256 hashes for each block of the file. The first hash is computed by extracting the bytes $[0, block\_size - 1]$ from the source file, and computing the SHA256 hash for this block, the next hash corresponds to the bytes $[block\_size, 2 * block\_size - 1]$ and so on.

The number of blocks (i.e. the number of hashes in the file) MUST match the computed number of blocks, using the formula:

$$blocks = \text{floor}\left(\frac{file\_length + block\_size - 1}{block\_size}\right).$$

As an example, a source file of 1000 bytes and a block size of 64 bytes yields 16 block hashes, each with a length of 32 bytes. As the header is 64 bytes the total length of the `cascade` file is $64 + 32 * 16 = 576$ bytes.

Once a file is downloaded entirely, it should have the size mentioned in the header and a SHA256 hash corresponding to the value in the header.

## 3.2 Tracker API

The tracker keeps information about which clients are currently active in the network (as in: it tracks the peers).

A client MAY request a list of peers currently serving a particular file by sending a request to the tracker.

The request header MUST look like the following:

```
4 bytes - "CASC", string literal
4 bytes - Protocol version, must be 1, unsigned integer in
          network byte order
4 bytes - Command, unsigned integer in network byte-order
4 bytes - Length of request data, excluding this header,
          unsigned integer in network byte-order
```

The tracker only supports two commands and the command MUST be one of these:

```
1 - List peers
2 - Subscribe peer
```

For either command `1` or `2`, the request length MUST be $32 + 4 + 2 = 38$. The data in the request MUST be in the following format:

```
32 bytes - The cascade hash
4 bytes - The peer listening IP
2 bytes - The peer listening port
```

The value for "cascade hash" is the SHA256 hash of the `.cascade` file and should not be confused with the SHA256 hash of the target file, nor the SHA256 hash of each block. To obtain the SHA256 hash value one can use the `shasum` tool found on most operating systems:

```
shasum - A 256 source.cascade
```

For either command, the tracker MUST respond the same, and the response header MUST have the format:

```
1 byte - Status code
4 bytes - Length of response, excluding this header, unsigned
          integer in network byte-order
```

The body of the response is always either a list of peers currently known by the tracker, or an error message. The makeup of this body depends on the 'Status code', and will be of the length given by the 'Response length'.

If there is an error, the status code MUST be `non-zero` and the response data MUST be a human-readable error message of the length provided.

If the response is valid, the status code MUST be `zero` and the response MUST be a sequence of entries of the following format:

```
4 bytes - Peer IP
2 bytes - Peer port
4 bytes - Last seen timestamp, seconds since UNIX epoch,
          unsigned integer in network byte-order
1 byte - "Good peer" flag, 1 meaning a course-hosted client
1 byte - Reserved, should be zero
```

This means that each record is 12 bytes and the number of records returned can be computed by dividing the response length by 12.

*Note:* The tracker may return peers in random order, and may return only a subset of all peers. Repeated requests MAY return different lists.

## 3.3 Client protocol

Once a client has obtained one or more peers that serve the desired file, they CAN request individual blocks from the file. The request sent to a client MUST follow the layout:

```
8 bytes - "CASCADE1", a string literal
16 bytes - Reserved, SHOULD be zero
8 bytes - The block number, unsigned integer in network
          byte-order
32 bytes - The cascade hash
```

The client receiving a request MUST return a response, with a header in the following format:

```
1 byte - Status code
8 bytes - Response length, excluding this header
```

The response body MUST be either an error message, or the data contained in the requested block. The makeup of this body depends on the 'Status code', and will be of the length given by the 'Response length'.

If there is an error, the status code MUST be `non-zero` and the response data MUST be a human-readable error message of the length provided.

The following error codes are currently defined:

```
0 - Success
1 - Invalid hash (i.e. client does not have the file)
2 - Block not present (i.e. client has not downloaded it yet)
3 - Block number too large (i.e. client has requested a block
    that does not exist)
4 - Failed to parse request
```

The client SHOULD use the appropriate status codes to report error conditions.

If the client has the requested block it MUST respond with the status code `zero` and the length of the data. The length of the data MUST be the length of a block, unless the block is the last in the file, in which case it MAY be smaller. The data following the length MUST be the data from the block.

After sending a response, the peer MAY close the connection. If the peer supports multiple requests on a single connection, it MAY keep the connection open and serve the next request. If the status code indicates a problem with the request (i.e. is not zero or 2), the peer SHOULD close the connection as additional requests are likely to produce errors as well.

## 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:

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

**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.**