# A6: Computer Networking (I)

Computer Systems 2019
Department of Computer Science
University of Copenhagen

Vivek Shah
Adapted by Anders Holst and Michael Kirkedal Thomsen

**Due:** Sunday, 15th of December, 10:00
**Version 2** (December 3, 2019)

---

This is the seventh assignment in the course on Computer Systems 2018 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 A7. 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 distributed chat service employing a combination of both client-server and peer-to-peer architectures using socket programming in C. The implementation task of the complete chat service would be spread over this assignment and the next one (A7). In this assignment, the programming effort relies solely on building the client-server 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 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].

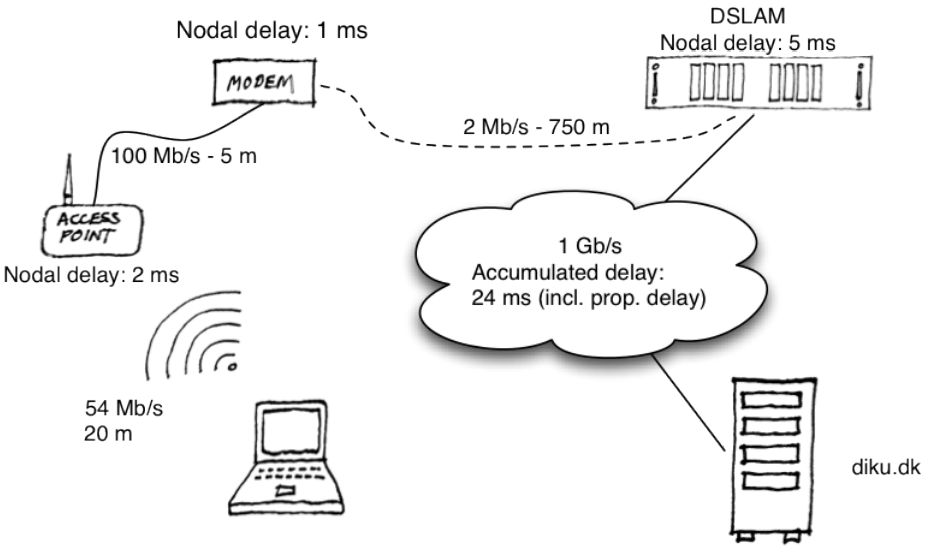


**Figure 1:** A typical DSL setup

[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 the client-server portion of the distributed chat service.

## 2.1 Design and overview

The distributed chat service comprises a peer client (`peer.c`) and a chat name server (`name_server.c`). Through a peer client, users log into the name server with their username and password. The server allows users to lookup other online users, send messages and read incoming messages. Eventually, the chat service will allow peers to connect and chat with each other independently of the name server.

For this assignment, you are going to implement functionality in the peer client and name server for logging in and out, looking up users, and for the peer client, safe exiting of the program. *We leave the actual messaging for A7.*

To build a bridge between implementation and theory, you are going to design and implement *your own* protocol, dictating communication between name server and client/s.

## 2.2 API and Functionality

### 2.2.1 Peer-side

The peer client uses a simple IRC[4]-like command functionality. The client should allow the user to perform the following actions:

2.1. Login - the user should be able to request logins to the name server, specifying a username, password, and her address information i.e., the IP address and port on which the client is going to be listening for name server respones (and later, peer chat messages). The syntax for the command is:

```
/login <username> <password> <IP> <Port>
```

2.2. Lookup - the user should be able to query the name server for the address information of a given username, or, if no username is provided to the command, of all users currently signed in. The syntax for the command is either:

```
/lookup <nick>
```

or, if the client wishes to see all online users

```
/lookup
```

2.3. Logout - the user should be able to log out of the name server. The syntax for the command is:

```
/logout
```

[4] https://en.wikipedia.org/wiki/Internet_Relay_Chat

```
$ ./peer 10.68.110.187 4242
/login aemylis topsecret 10.68.110.189 1337
>> Welcome to the name server.
/lookup aemylis
>> sortraev is online.
>> IP: 10.68.110.189
>> Port: 1337
/lookup sortraev
>> sortraev is not online (or username invalid).
/lookup
>> 1 user online. The list follows:
>> User: aemylis
>> IP: 10.68.110.189
>> Port: 1337
/logout
>> Goodbye ...
/exit
>> Closing client ...
$
```

**Figure 2:** Sample client interaction

2.4. Exit - terminate the client program. If client is logged in at exit; request a logout first.

```
/exit
```

A sample interaction with the peer client program is shown in Figure 2. In the example, lines prefixed with ">> " are responses from the name server. *Please note that you are free to design your own response messages, long as they are meaningful; this is part of the protocol you devise.*

### 2.2.2 Server-side

The server should set up needed sockets at start-up and immediately start accepting client connections.

Record should be kept of usernames and corresponding passwords of clients who are allowed to login to the name server (for this, we present the template `struct client_t` in `name_server.h`, which you should extend with any fields you find necessary, and `client_t *clients[]` in `name_server.c`, which can store the records).

Upon successful connection and login of a client, the server should delegate servicing the client and continue listening for new clients. It is up to you to choose how client sessions should be handled based in the three different approaches given in the lectures; we recommend that you choose either the multi-threaded or multi-processed model (as this should feel familiar after A4 and A5), in which each new client is served by a separate thread or process,

but you may also choose the event-driven model as presented in BOH[5] (Chapter 12.2).

With the client session up and running, the server should continuously receive and satisfy commands from the client until it chooses to log out of the name server. Below is a list of commands which the name server should support, along with the semantics of the name server for each:

2.1. Login - Clients presenting a known username and password combination, as well as valid address info, should be able to log into the name server. Upon login, the given info should be tied to the user session.

2.2. Lookup - Clients should be able to query username and address info of other users logged in. If a specific username is given, only this user should be looked up; else all online users should be listed.

2.3. Logout - At any time, clients should be allowed to log out of the name server, regardless of whether they have unread messages waiting or not.

#### 2.2.3 Communication protocol

At this point you may be wondering how clients and the name server should communicate. We have only specified the API's of the peer and name server programs; it is up to you to design the communication protocol, which supports the API between client and server!
Some interesting things you should definitely consider are:

2.1. How should messages and requests be formatted? Should they follow a common template, or should each command have its own format when transmitted? Are messages sent in segments or are they batched? (this is for example relevant in the case of the Lookup command)

2.2. Should the name server and client expect and wait for acknowledgement from one another, for example in relation to logins and logouts? How does client and server notify each other of errors and miscommunication?

2.3. Upon receiving requests from users, should the name server verify the validity of the command, or simply assume that the command has been sanitized client-side and no information lost in transmission?

2.4. Which transport protocol would be suitable and why?

In designing your protocol, you may be able to take inspiration from some of the protocols you have seen in the course thus far.

[5]Computer Systems: A Programmer's Perspective, Randal E. Bryant and David R. O'Hallaron, Pearson, 3rd and Global Edition

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

- `peer.c` contains the client. At the time of hand-out, the client program supports user input through stdin, as well as parsing of commands and arguments. A switch-statement chooses the appropriate action to take based on the current command, but for now, the switch cases are only dummies. The code contains a total of six `TODO`'s, and it is your job to finish the implementation by filling out the code missing under each `TODO` in `peer.c`. Each `TODO` is accompanied by one or more hints to get you started.

  *If you finish all six* `TODO`*'s, you should have a fully functioning client program! :):)*

- `name_server.c` contains the name server. As you will notice, this file is a little sparse on hand-out compared to `peer.c`; here, you will only receive a couple of hints in the form of in-code `TODO`'s. You must figure out the rest on your own - simply implementing the `TODO`'s will *not* give you a fully functional name server (but you are free to take inspiration from course material, such as the echo server discussed in the lectures and Chapter 12.3 of BOH).

- `peer.h` and `name_server.h` each contain peer- and server-specific macros and function prototypes, while the latter contains the mentioned template `struct user_t`.

- `common.c` and `common.h` contain macros and functions needed by both programs, including functions for parsing IP addresses, port numbers, and client commands and arguments. We encourage you to take a look at this file and add to it as you see fit.

- As usual, we provide `csapp.c` and a `Makefile`.

## 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 that each API functionality implemented works, and are able to document those which do not. However, we can recommend that you use "<" to guide input for stdin (instead of copy-paste), as done in the testscript for A3.

Simply running the program, emulating regular user behaviour and making sure to test each command should suffice, but remember to note your results.

If you only manage to implement the peer, you can make some basic tests toward the echo-server.

*Note: test your implementation by first running the name server and peer client on the same machine before trying to connect multiple different machines, since some*

*networks disable certain forms of traffic that can affect your testing procedure. Especially, on Eduroam you cannot make communication between to computers; e.g. make a hotspot with your phone.*

## 2.5 Recommended implementation progress

As mentioned in the previous section, `peer.c` contains six `TODO`'s - the satisfaction of which should yield a functioning peer client - while `name_server.c` will require you to work more independently. 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. Devise your communication protocol for the peer client and name server.

2.2. Finish all six coding `TODO`'s given in `peer.c`, such that the client program adheres to the given API.

2.3. Extend the `struct client_t` in `name_server.h` and start implementing `name_server.c` as per the API given for the name server.

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

2.5. 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:

- Document the protocol you designed, including message format and exchange protocol. Give a small discussion (is it efficient, does it provide any service guarantees?). (Approx. weight: 10 %)
- Document technical implementation you made for the peer - cover in short each of the six `TODO`'s for `peer.c`. (Approx. weight: 15 %)
- Document technical implementation you made for the server - that is additions you make to `name_server.c` which you find relevant.

  For example, which model did you choose for the server (multi-threaded, multi-processed, or event-driven), and how do you read/write to/from sockets? (Approx. weight: 30 %)
- Discuss testing and shortcomings (if any) of your implementation, and how these might be fixed. It is not necessarily expected of you to build a fully functional chat service, but it *is* expected of you to reflect on the project. (Approx. weight: 15 %)

- The current design of the distributed chat service uses a centralized name server. Is this a good idea? What advantages and disadvantages are there of running a centralized name server? How could we change the service so that the name server would be distributed? (Approx. weight: 5%)

*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

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).

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.**