FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Assignment 6: Computer Networking-1

CompSys 2017
Department of Computer Science
University of Copenhagen

Peter, Tobias & Xueying

17. December 2017

1 Theoretical part

1.1 Store and forward

1.1.1 Processing and delay

Besides **propagation delay** (*"the time it takes for the bits to travel across the wire at near the speed of light"*[1]), packet switched networks may suffer from **nodal processing delay** (delay caused by a device having to processes/examines the packets), **queuing delay** (the wait time it takes to be transmitted onto the link), and **transmission delay** (the amount of time required to transmit all of the packet's bits to the link). The aforementioned types of delay make up the **total nodal delay**

1.1.2 Transmission speed

Part one: To calculate the RTT (the time it takes for a signal to travel from client to server and back), we need to ascertain the following: Firstly, the physical distance between the laptop and the DSLAM. Secondly, the delay at every link. The physical distance is calculated as follows:

- Distance - laptop and access point: 20 m
- Distance - access point and modem: 5 m
- Distance - modem and DSLAM point: 750 m
- Total distance: 775 m

Since the propagation delay in all links visible is $2.4 * 10^8 m/s$. The propagation delay between the laptop and DSLAM may be calculated as follows:

$$\frac{775m}{2.4 \cdot 10^8 m/s} = 0.00000322916s = 0.00322916ms$$

The propagation delay from the laptop to the DSLAM is therefore 0.0032 ms. The total queueing delay at every link is calculated as follows:

- Node delay at access point: 2 ms
- Node delay at modem: 1 ms
- Node delay at DSLAM: 5 ms
- Accumulated delay at DSLAM and diku.dk: 24 ms
- Total delay: 37 ms.

The RTT may now be calculated:

$$2(37ms + 0.0032ms) = 74.0064ms$$

[1]Kurose and Ross (2013): *Computer Networking*, p. 24.

Part two: Since the 640 KB of data can be sent as a single package, the total transmission time may be calculated as follows:

$$\frac{0.640Mb}{54Mb/s} + \frac{0.640Mb}{100Mb/s} + \frac{0.640Mb}{2Mb/s} + \frac{0.640Mb}{1000Mb/s} + 0.074s = 0.41289s$$

The total transmission time will therefore be 0.41289s.

1.2 HTTP

1.2.1 HTTP semantics

Part one: The method field is part of the request line in a HTTP request message. The method field specifies what action the sender of the request wants to take - like asking the recipient to perform the specified method / action with the given input. One of several methods is the GET-method. GET is used to request an object, E.G. an index page (identified by the sender-provided address in the URL-field). Another method is the POST-method, which is similar to the GET-method in the sense that it is used to retrieve an object, though it uses the information, E.G. search terms in a web form, contained in the entity body of the HTTP request to determine what object to return to the original sender.

Part two: The Host-header field contains the URL address of the host from which the sender is requesting an object. This field must be included in case the client is indirectly accessing a host via a proxy server – requesting objects from the host via the proxy server that is. The proxy server may store frequently requested objects in a web cache and return these, but some objects may not be stored and then the proxy server will need the host address to retrieve the object.

1.2.2 HTTP Headers and fingerprinting

Part one: These header fields are used to add a user-session layer to an otherwise stateless HTTP server, thus enabling the server to provide content tailored to the user, E.G. by storing user-specific settings and search history. In practical terms, the Set-cookie-field is used in HTTP responses by servers to give the client a unique identification number, which is also stored in a back-end database along with user-associated information. Conversely, the cookie-field is used by the client in HTTP requests to identify itself with the server. Using the cookie, the host may access its database and retrieve the user's settings.

Part two: The ETag may work as a cookie, since it is stored by the client and sent to the host via HTTP requests. Upon receiving the ETag, the server may handle the tag as a cookie, I.E. store the tag in a database and associate it with some user sending requests. Then, rather than generating a new ETag for some updated resource and returning it, the host may just repeat the ETag sent by the client, which the client will then save. Upon sending another request for this resource, the client will be using the same ETag and the host will recognize the client.

1.3 Domain Name System

1.3.1 DNS provisions

The three most important goals of DNS3 are met in practice by different means:

- **Fault tolerance:** The fault tolerance lies in the fact that there exist multiple DNS servers, which means that even if one server fails, the system will be still be able to go on.
- **Scalability:** DNS are made out of multiple DNS servers, organized in a hierarchical fashion, this makes it possible for it to tackle the scalability problem.
- **Efficiency:** Since DNS makes use of DNS caching, this greatly increases perfomance and efficiency.

1.3.2 DNS lookup and format

Part 1: With CNAME type records, different host aliases might be associated with one canonical hostname, E.G. `tempelkoderne.dk` might be an alias for the (full) canonical hostname `relay.bornholm.tempelkoderne.dk`.
DNS may aid in distributing the load on several servers. Consider a server which serves many users at once, such as `skat.dk`. This site may be replicated on several servers to distribute the traffic, though the actual distribution is handled by DNS in the following way: The DNS database associates a list of IP-addresses (of the aforementioned replicated servers) with `skat.dk`. When a client requests the IP-address of the domain, DNS will randomly "rotate" this list (rearrange the order of the items) and return it to the client, who will usually send its request to the first address in the list. The load is thus distributed.

Part 2: As the name implies, an iterative lookup / request / query asks the DNS server to iterate over connected servers, asking for some IP address, who then return either the IP or the address of some other DNS server, who may know the IP of the specified domain. A recursive lookup / request asks the DNS server to forward the request to other DNS servers, until the IP of the specified domain is found. The IP is then sent back to the client along the request chain.
Recursive lookups – although unsafe and potentially very resource-demanding, if the request is passed on through many servers – are justified by DNS Caching. Specific / local servers may cache domain-IP address pairs that are requested often. When a request reaches such a server, it may quickly return the IP-address of the domain. Because of this, one avoids waiting for root, TLD and authorative servers to process an iterative request.

2 Programming Part

2.1 Compiling and running the code

The chat client library and test programs may be compiled and tested via a UNIX-like command line in the following way:

1. Unzip `src.zip` and navigate to `.../src/`.
2. Compile the library and programs by running this command:
   `make`
3. Run the `name_server` program in one command line using this command:
   `./name_server`
4. Run the `peer` program in another command line using this command:
   `./peer`

The user will now be able to interact with the `name_server` via the `peer`-client using the following commands:

- `/login <nick> <password> <IP> <Port>` : the user must first log into the server.
- `/exit` : the user may exit the `peer`-program if they are not logged in to the `name_server`.
- `/logout` : the user may logout, if they are logged into the server.
- `/lookup` : the user may request a list of all other online users, if the requesting user is logged into the server.
- `/lookup <nick>` : the user may request the status and information of a specific user via their *nick*, if the requesting user is logged into the server.

2.2 Testing

In order to test our implementation, we designed various use-case scenarios to test the core functionality and some border cases. This is to ensure that 1) the program works as intended and 2) the program has a satisfying degree of error handling. Please see the appendix for an overview of these scenarios.

In conclusion, we are satisfied with the functionality of our program, disregarding some minor print-formatting flaws. The client and server seem fairly robust although we would like them to handle bad input in a safer manner, I.E. when the server receives a login-request with more arguments than specified in the documentation and the server still processes the request.

2.3 Implementation

The peer client and name_server are both based on the multi-threaded echo_server example given in *Computer Systems - A Programmer's Perspective*. Thus they are both continuously writing and listening on the established connection.

For the sake of keeping things simple and human-readable, we decided to send commands / messages back and forth as they were; as char arrays. All command parsing is then done by the recipient, mainly the server, as the server should have the main responsibility. When handling more complex commands, such as `/login`, which takes several arguments, we decided to let the server use `strtok()` and tokens to cycle through the arguments and act accordingly. In another case, when the peer checks if it has been logged in or out, we simply use `strcmp()` on a whole char array.

2.4 The Peer Client

When the user starts the peer program, there are two available commands: /login or /exit. Using a `while`-loop, the program accepts only these two commands as input - otherwise it prints an error message.

2.4.1 `login()`

The main purpose here is to communicate with the server, such as sending the user input to the server and receiving the server's replies.
This function will first connect to the server to perform a special handshake - in other words: attempt to log in. If the server does not affirm the login request, the client will disconnect. If the request is affirmed, the client remains connected. The `peer`-client will then write to and read from the common file descriptor until a logout request is affirmed, in which case it breaks out of the `while`-loop and terminates the connection. While connected, the user has access to the various lookup-features.

2.4.2 `process_reply()`

The `process_reply` function serves the purpose of checking if the server has affirmed the logout-request in which case the function will return -1 and otherwise 0.

2.5 The Name Server

As big as our current user list is, our current setup is an executable solution, using a centralized name server. The user data is stored on one server and hence the communication is local. User information is easily accessible, since everything is stored on one server. However, if our server should serve hundreds or thousands of users, a centralized name server would not be able to handle the load. It would become slower or even crash, due to the heavy traffic. A solution might be to distribute the database across several servers, thought this requires some coordination. We might draw inspiration from the theory mentioned above: we may have a root server who is responsible for distributing traffic to different host servers who coordinate their separate databases. Though a DNS-like system, they may forward lookup-requests for users they didn't find in their own database. In this fashion we may distribute our name server.

2.5.1 `echo()`

Since our implementation is based on the echo-server, we decided to simply extend the echo-function. It now continuously reads anything written to the file descriptor, sends the input to a helper function – `process_cmd` – to be parsed, and writes back a reply to the file descriptor. Once the `peer` disconnects, the status of the logged out user is updated.

2.6 `struct list`

We decided to use a struct list for storing the mock users used for testing our program.

2.7 `process_cmd()`

The `process_cmd`-function in name_server does the initial parsing, before calling appropriate helper functions according to the user input. If the input is invalid, the function returns an error message.

2.7.1 `login()`

When user using the login command, the corresponding `login` function will first check whether the given user is already logged in, if not, it will proceed with the user name and password, to verify them. User will receiver a reply, either the given information checks out or not. This reply will be a signal to the `login` function in peer.

2.7.2 `logout()`

The `logout` function will send a reply to the peer, where a disconnection will triggered. When there is no longer a connection between the peer and the server, the logged_in value of the user will be set to 0.

2.7.3 `lookup()`

If the user does not include any nicknames when they use the `/lookup` the `name_server` will print out the number of online people, and information of all the people that are online. It does this by using two for-loops one, for checking how many people that are online, and one for sending the information of the people that are online back. Having two for-loops may be seen as a waste of resources, however the design-choice for this, is based upon the fact, that we wanted to print out the number of people that are online first.

2.7.4 `lookup_nick()`

When a user is looking for a specific user, a for-loop will try to find a match in the user_list. Once there is a match, the information of it will be passed to reply, either the matching user is online or not. If there is no match, a "not found" reply will also be passed.

3 Appendix - Testing Scenarios

In all scenarios, the server and client are started first. Below are listed input-commands and the expected effect.
Exit-command

Input:	Effect:
/exit	peer-client terminates.

Two user login and lookup

Client:	Input:	Effect:
A	/login fisk 654 127.0.0.1 8080	user logged in.
B	/login banana 123 127.0.0.1 8181	user logged in.
A	/lookup	list of online users printed; fisk, banan.
A	/lookup pear	error message printed.
A	/lookup banana	information on banana printed.
A	/logout	user logged out.
A	/exit	peer-client terminates.
B	/logout	user logged out.
B	/exit	peer-client terminates.

Invalid command

Input:	Effect:
/login fisk 654 127.0.0.1 8080	user logged in.
hej	error message printed.
/logout	user logged out.
/exit	peer-client terminates.

Wrong username

Input:	Effect:
/login hej 654 127.0.0.1 8080	error message printed.
/exit	peer-client terminates.

Wrong password

Input:	Effect:
/login fisk 123 127.0.0.1 8080	error message printed.
/exit	peer-client terminates.

Too many arguments in login request still works

Input:	Effect:
/login fisk 654 127.0.0.1 8080 123	user logged in.
/logout	user logged out.
/exit	peer-client terminates.