A3 - CompSys

Nikolaj Krarup, ltf688,
Simon Wissendorff Seheim, fzt420
Emil Vedel Thage, prn150

September 2023

Introduction

In this short report we will start by going over the theoretical questions from the assignment. Afterwards we will discuss our implementation of the A3 assignment. We have been tasked with creating a client which is able to register a user and request files from the server.

Theoretical Part

Store and Forward

Processing and delay

Although that is true, there are more than one reason delays happens, all the reasons are called `the total nodal delay`, consisting of `nodal processing delay`, `queuing delay`, `transmission delay` amongst other delays. `Processing delay` is the time it takes to determine where to direct a packet. `Queueing delay` is the time it takes to for the packet to be transmitted onto the link, this time will depend on how many other packets there are waiting. Transmission dealy is the time it takes to transmit all of the packet's bit onto the link.

Transmission speed

Part 1
To calculate the total round trip time RTT, we need to calculate the time it takes for a packet to travel from the client to the server and back to the client. The RTT includes propagation delays, queuing delays and nodal processing delays. We can sum these delays together, and mulitply them by 2 to get the total round trip time. Starting with the propagation delay, this is the time it takes for the signal to travel between links. As stated the propagation speed is $2.4 * 10^8$ m/s. So going by the length between the links we can calculate the propagation delay, we leave out the delay between the DSLAM and the server as the accumulated delay is stated:

$$d_{prop} = \frac{20 \text{ m}}{(2.4 * 10^8 \text{ m/s})} + \frac{5 \text{ m}}{(2.4 * 10^8 \text{ m/s})} + \frac{750 \text{ m}}{(2.4 * 10^8 \text{ m/s})} \approx 3.2 * 10^{-3} \text{ ms}$$

Since the total propagation delay is so small it could be leaved out, but since we have calculated it we will use it for the total RTT. Now for the queuing delays and nodal processing delay, the queuing delay is stated to be contained in the nodal delays, so we can simply add these together

$$d_{proc} + d_{queue} = 2 \text{ ms} + 1 \text{ ms} + 5 \text{ ms} = 8 \text{ ms}$$

The transmission delay is not included in the round trip time, as the RTT is the speed of a small package going back and fourth between the client and server. For example the handshake protocol before transmission of data, so since no large data transfer are included in the RTT, transmission delay is superfluous. Finally we need to add the accumulated delay between the DSLAM and the server, which is stated on figure 1, which includes propagation, nodal processing and queuing delays

$$d_{acc} = 24 \text{ ms}$$

We can now add our delays together to get the RTT, we will multiply it by 2, since it is the time back and forth between client and server

$$RTT = (d_{prop} + d_{proc} + d_{queue} + d_{acc}) * 2 = 64.0064 \text{ ms} \approx 64 \text{ ms}$$

So if we leave out the small propagation delay, RTT is 64 ms.

Part 2
To calculate the total transmission time we need to calculate the transmission delay of 640 KB data, and add it to the RTT. We add it to the RTT since this will include the delays of sending the data to the server (excluding transmission delay), as well as the small response from the server. To calculate the transmission delay we need the data size in bits L, and the transmission rate between the devices in bits/sec R. So we need to change everything to bits, it is unspecified whether its stored in binary or decimal based system, so we will use the decimal based system for easier calculations. Giving us $L = 640 * 1000 * 8bits = 5.120.000bits$ and calculating R for the individual links. Now using the formula for transmission delay we get

$$\begin{aligned} d_{trans} &= \frac{5.120.000b}{432.000.000b/s} + \frac{5.120.000b}{800.000.000b/s} + \frac{5.120.000b}{16.000.000b/s} + \frac{5.120.000b}{1.000.000.000b/s} \\ &= \frac{28972}{84375}s \approx 343 \text{ ms} \end{aligned}$$

Now adding this to our RTT we get

$$T_{trans} = RTT + d_{trans} = 407 \text{ ms}$$

HTTP

HTTP semantics

Part 1
The method field in the request specifies what action is to be done. The `GET` is used to request an object identified in the `URL` field. The `POST` method on the other hand is used to once again request an object but the objects content depend on the users

input in the form fields. When requesting a web page through the `POST` method the entity body will then contain the input from the form fields.

Part 2

Why is a header necessary? It is used to specify on which host the object resides.

HTTP headers and fingerprints

Part 1

When accessing a website for the first time the request from the user is met with an respond containing a unique identification number from the web server telling the users browser to append a line to a special cookie file it manages (different numbers for each web service it visits). When the identification number has been set, it is then put into the future HTTP requests as a cookie header line. This way the web server will know when the user with the given requests different sites and is therefore able to collect data on their behaviors.

Part 2

The `ETags` can be used just like with both `If-Match` as well as `If-None-Match` to tag entities. When tagged the entities can be tracked in the same way as cookies. When tagged it is possible for the method to check if the tagged entity is up to date or if it's one specific one being checked for just like the web server being able to check the users browser.

Domain Name System

DNS provisions

To ensure security as well as scalability a hierarchical structure is implemented. This way if one DNS server fails not everything is lost since every layer has more than one server. In the same way if a section needs to be expanded more servers can be added to a layer. This layered structure also creates effiency since one server doesn't have to all others to find a request but rather go through the layers in a hierarchical (up or down) order to ensure minimum wasted time

DNS lookup and format

Part 1

A CNAME type record makes it possible for a client to lookup a mnemonic name for a site which has been linked up to the canonical name of set site. One way DNS balances workload is by using DNSA caching to store (usually for a maximum of two days) ip addresses being requested. Another way is by using local DNS servers which the user uses to communicate with the root DNS, where the root gives information on where on TLD server the request might be found. The local then communicates with the TLD about what(and where) the IP address is to be found. (Caching

should reduce the amount of times the process takes place) Part 2
When using an iterative lookup the users device communicates with firstly the root server, which then tells the client what TLD server to find it on. The client then goes and talks to the TLD servers and find out which other server it might be on. Each and every step is communicated between the client and DNS server in contrast to the recursive. The recursive only communicates once with the root server whereas the root server then communicates with the layers below until it is found. This is usually faster since every layer gets to cache the searched name. Whereas the iterative only cashes on the found layer, the recursive cashes on every layer it passes.

Implementation

get_signature

Our implementation of get_signature was largely taken from a practice class implementation with a couple of changes. It concats the salt together with the password to then pass it onto get_data_sha which then creates the hashed and salted password.

register_user

When formatting a request to send to the server we have followed the format defined in the assignment, using the structs from `networking.h`. To make the code more neat we have created a helper function `build_request()` to fill out the struct, making use of the `get_signature()` function to get the salted and hashed password. The `get_signature()` functions leaves the payload empty when given the length of 0, following the format of registering a user. The header is written to the server using the `compsys_helper_writen()` trough the network socket. To read the response a compsys helper state has been initialised globally in the program, making it possible to use `compsys_helper_readnb()` to read the incoming response header. The response header is then analysed, and if additional bytes besides the header is sent, they are read and printed to the terminal. Using a salted and hashed password increases security.....

To save the client's salt between sessions we have used a .csv file to store the salts locally. This way when a user is logging on to the server from the same machine the salt will be applied. This might not be the most safe solution as the file is not encrypted, but since it is only stored locally it shouldn't be a big problem.

get_file

We used the same way of formatting the request as in register_file except with a proper length and payload. The length is converted to network byte order and the

payload set to the file name.
After having written the request to the socket, the handling of the response begins. To handle the file replies, we've made a struct RespHeader to contain a block's information with all relevant fields the block contains.
We've made our get_file able to handle all block sizes in one big implementation. If the status code == 1 and there are more than 0 blocks, it gets to work. We've chosen to make an array of blockCount size to contain all blocks' payloads. A loop then orders the blocks' payloads inside the array according to their block number. During each iteration, the recieved and the expected block hashes are compared to make sure nothing has gone wrong during transmission.
Then after the ordering, a loop fprints every index from 0 to blockCount into a file named after to_get and the hash of the file is compared to the expected hash to make sure the file has been put correctly together. If the response can't be processed as a file, the payload will be printed which is most likely an error message.

main

The main function handles the user interaction and calls to our implemented functions. Here we also establish connection to the network socket using the compsys helper function. Building upon the handed out code we have added a way for user's to login to their already registered accounts. To do so we check whether the username is listed in the .csv file, if it is, we apply the salt from the .csv file instead of generating a new random salt. If the user is registered we skip the call to `register_user()` and go straight to the `get_file()` function.

We've made a never ending while-loop that creates user-interaction. It will constantly ask for files to make a request for until you exit by writing quit.

Testing

We've largely been testing our implementation with manual testing during development. We've only been testing with 3 files: tiny.txt, hamlet.txt and a non-existing file.
By running it with tiny.txt and hamlet.txt, we observe a file appearing in the directory of the same name, and since we check the hashes during assembling, we know it to be the correct file. Additionally, a it will be written to the terminal what was received. Whenever you use a invalid file, it will simply print the response from the server, most likely an error message. We've also called the different files in different orders to make sure the calls don't interfere with each other.

Limitations and Potential Problems

Whenever we close the connection to the server, the server reports receiving a bad request. We're not sure what's causing it since we're not sending the server any requests and the python code doesn't enlighten us.

Conclusion

This assignment has given us a lot of insight in communicating between a server and client. Especially the importance of upholding the networking protocols, like the formats of the request and response headers. We have also learnt how to manage data being sent over multiple packages, and the security layer of salts and hashes.