Assignment 3 — Computer Networking (I)

Schmidt, Victor Alexander, `rqc908`
Ibsen, Helga Rykov, `mcv462`
Barchager, Thomas Haulik, `jxg170`

Sunday 16:00, November 20th

1 Theoretical Part

1.1 Store and Forward

1.1.1 Processing and delay

Before the router can send the packet to the outbound link, it needs to examine the packet's header and determine where to direct the packet. This is known as the **nodal processing delay**. Another reason for delay is related to the mechanics of the output buffer. It stores packets that the router is about to transmit onto the outbound link towards the destination. If the link is busy transmitting another packet, the newly arriving packets must wait in the output buffer until the outbound link is available. As a result, that leads to output buffer **queuing delays**.

1.1.2 Transmission speed

Part 1: RTT is defined as the time it takes for a small packet to travel from client to server and back to client. It includes queuing delay and packet processing delay (here Tn), as well as propagation delay (Tp), except transmission delay.

$$RTT = 2 \cdot Tp \tag{1}$$

We find propagation delay (Tp) as:

$$Tp = Tp_1 + Tp_2 ... + Tp_n \tag{2}$$

Each Tp is a quotient of the distance between the two nodes/routers and the propagation speed, i.e. $Tp = \frac{d}{s}$. Since RTT also includes the queuing delay specified for each node (Tn) in Figure 1, we have the total Tp as follows:

$$\begin{aligned} Tp &= (Tp_1 + Tn_1) + (Tp_2 + Tn_2) ... + (Tp_n + Tn_n) \\ &= (\frac{20}{2.4 \cdot 10^8} + 0.002) + (\frac{5}{2.4 \cdot 10^8} + 0.001) + (\frac{750}{2.4 \cdot 10^8} + 0.005) + 0.024 \\ &= 0.002 + 0.001 + 0.005 + 0.024 = 0.032 \end{aligned}$$

We insert the result into (1) and get the total RRT:

$$RTT = 2 \cdot 0.032\text{s} \approx 0.064\text{s} = 64\text{ms}$$

Part 2: To find the total transmission time, we have to add the RTT calculated above and the transmission delay, which is the quotient of the length of the packet (L bits) and the transmission rate of the link from router A to router B (R bits/sec). We can find the total end-to-end delay of sending 640 KB of data from the client to `diku.dk` webserver over a path consisting of N links each of rate R as:

$$\begin{aligned} d_{end-to-end} &= N \cdot \frac{L}{R} \\ &= 3 \cdot \frac{0.08\text{Mb}}{54\text{Mb} + 100\text{Mb} + 2\text{Mb} + 125\text{Mb}} \\ &= 0.00085\text{s} = 0.85\text{ms} \end{aligned}$$

The total transmission time can now be found as:

$$\begin{aligned} TTT &= RTT + d_{end-to-end} \\ &= 64\text{ms} + 0.85\text{ms} = 64.85\text{ms} \end{aligned}$$

1.2 HTTP

1.2.1 HTTP semantics

Part 1 The purpose of the method field in a HTTP request, is to indicate what type of action you want to perform on any given resource. The **GET** method is used when you only want to retrieve data. **POST** is used when you want to send data to a server to create or update a resource.

Part 2 The `host` header is used to specify the domain a client wants to access. It is necessary because some servers host more websites and applications at the same IP, and without the host header they will not know where to direct the request.

1.2.2 HTTP headers and fingerprinting

Part 1 HTTP cookies is a way for any server to store and retrieve state information from any client application. Cookies is a small piece of data (basically a serial number or a user ID generated by the server and saved in its buffer) that a server sends to a client. The client then stores these cookies (her own user ID generated by the server) locally and sends them back to the server with subsequent requests. Cookies are mainly used for e.g. Session management, personalization and tracking.

Part 2 ETags is a way for a server to know that a client has the most up-to-date-version of the files on that server. If the client does not have the most recent resources from the server, it will tell the client to fetch those resources and cache them. Like cookies, eTags are stored in cache by the client and are sent back and forth between client and server to determine whether the client has the most recent version stored in its cache.

1.3 Domain Name System

1.3.1 DNS provisions

DNS ensures fault tolerance, scalability and efficiency through load balancing. It works by distributing traffic across several servers instead of a single one. In practice, it does this by providing different IP addresses in response to DNS queries. The most common DNS load balancing is called round-robin DNS.

1.3.2 DNS lookup and format

Part 1 CNAME records is a type of resource that maps one domain name to another. This type of records is useful when a server has multiple services running on the same IP address. The advantage of using CNAME is that if the IP address has been changed on the server, then any CNAME record pointing to

that host will also be changed.

Part 2 The difference between recursive and iterative lookups is when one DNS server (called recursive resolver) communicates with several others (called authoritative servers) to locate an IP address and return it to the client.

In case with the iterative DNS lookup, the client (called stub resolver) communicates directly with each DNS server involved in the search for the IP address in question.

Recursive lookups tend to be faster than the iterative ones because the recursive resolver doesn't have to send requests to each and every authoritative server back and forth (iteratively) until one of those actually has the wanted ID address, i.e. recursive lookups have less RRT than iterative ones. And when the requests are send in the recursive order from the recursive resolver to one of the authoritative servers, which then redirects the request to the next authoritative server and so on so forth until one of the authoritative servers does possess the desired ID address, which is then passed back to the client along the same route. This is especially useful when a DNS server serves a lot of clients.

2 Programming Part

2.1 Protocol

To program our solution we were given a protocol to follow. We used this protocol as a guideline to create our implementation. Considering the security side, we had to handle the issue of validating our user with a username and a password. A password should never be sent over to a server in plain text. Therefore, we needed to hash our password turning it into a "long string of random numbers", which is much harder to crack. This architecture can yet be vulnerable to dictionary attacks and brute forcing — so it is necessary to add a bit of salt to the password as well, prior to hashing the whole thing altogether. By combining the password with a randomly generated salt and then hashing it, we are able to make it terribly more difficult to decode, so it will be a complete waste of time for a hacker trying to crack it.
Even though this is a pretty secured way of dealing with sensitive data, it still has a few flaws: e.g. the hashed password and the salt still need to be stored somewhere. Which means that if a hacker gets access to the server's database, containing the hashed password and the salt that was used, the salt would be meaningless because the hacker would just need to add the salt to his precomputed hashes in the rainbow-table. To improve that flaw, an expansion of the protocol is needed that specifies in particular how the salt and hashed password should be handled and stored.

2.2 implementation

Small prelude: we have implemented the client in response to the version of the server which does **not** shuffle the order of the sent file blocks.

Creating the signature: is done by copying the password and salt to the same character array and then the new array is hashed. The hash is the signature.

Register user: first makes use of the `get_signature` method. The rest of the implementation is quite trivial, so we will briefly cover each step:

1. after creating the signature;
2. the request header gets built inside the buffer;
3. (since there is no payload length when registering a user) we send the request to the server and read/print the response.

Getting a small file and a large file: is handled by the same function. It creates a new file to write to, creates the signature, builds the request header and body and then handles the server response, but only if the server response is `STATUS_OK`. It handles the server response, by assembling the payload length and then writing to the previously opened file with the line `fwrite( &buffer[RESPONSE_HEADER_LEN], 1, payload_length, f);`. In short, this line writes n elements from the start of the response payload/body, where $n =$

`payload_length` to the file `f`.
As previously stated, please note that our implementation does not take the order of the blocks into account.

Building the request header: have been done using a function, since the request header is always expected to have the same length. This is done by copying every bit of the username, signature and request length into a given buffer.

Interpreting the response information: when we receive it in the buffer, we read it as a `char` or `uint8_t` (8 bytes). However, the first four entries together (read as binary) would give us the actual length of e. g. the payload. Therefore we have created a helper function, which converts 4 `uint8_t` to 1 `uint32_t`, by shifting the bits to the correct order.

Implementation of tests in main: ... will be covered in the test section.

2.3 Tests

What was tested?

As mentioned in the assignment, we have made the following test-cases:

1. Can I register a user? (expected: yes)
2. Can I register a user, with an empty username? (expected: no)
3. Can I register the same user twice? (expected: no)
4. As a user, can I get a small file? (expected: yes)
5. As a user, can I get a big file? (expected: yes)
6. Is it possible to get a file, without being a registered user? (expected: no)
7. Can I get a file, which I know does not exist? (expected: no)
8. Can I get an empty file? (expected: no)

This was tested on a Windows computer, imperatively in the `main` method. It is important to note, that the ”expected” outcome is defined by ourselves, in other words, the response we find appropriate.

The only test that partly fails, is test # 2. The reason it only partly fails, is because the server responds, that the user can not be created (which is correct), but the server response is of the wrong response payload.

The whole response: `(Server response) Cannot register user under name 'tiny.txt`

The expected response (or something like it): `(Server response) Cannot register empty username`

2.4 Shortcomings

2.4.1 Technical shortcomings

Checksum and payload hashing: In the server response, there is a checksum and a hash of the payload, which is supposed to make sure each response body and the whole payload contains the correct information. However we do not compare this in our implementation, which is a clear shortcoming. The assignment did not demand this functionality either, but assuming something happened to the response body/payload during transportation between server and client, implementing a checksum-check and payload-hash-check could be vital to maintain correct data transport.

Assuming we would deal with a shuffled order of blocks: The client has not been built to support the reassembling of blocks into the correct order. Currently, each block payload is written down in order. A change like this should be trivial, but we simply ran out of time. Assuming the client is supposed to work in the payload order, this is definitely a shortcoming.

Handle too few response `status` messages: The current implementation only ever relates to the `status` message, when receiving a file from the server. In reality, the client should handle every `status` response, so this is a shortcoming on our end.

2.4.2 Other observed shortcomings

Password in plain text: Another possible shortcoming could be the fact that the password, as it is being typed, appears directly in the console as plain text, which we believe is a potential security issue. A solution to that could be hiding the plain text, so that it appears in the console in the form of, say, star symbols.