Computer Systemer - Assignment 6

vmf457

pmj428

spz377

tqw257

Team 2

December 15, 2019

1 Introduction

This assignment was divided into a theoretical section and a programming section. The theoretical section contained a list of theoretical questions, often including calculations. In the programming section, we were required to complete the implementation of a distributed chat service using both client-server and peer-to-peer architectures using socket programming in C. This assignment focused on building the client-server portion of the architecture.

2 Theoretical Part

2.1 Store and Forward

Processing and delay: Delays in packet switched networks are attributed to a few different kinds of delays. Processing delay, which refers to the time taken by the CPU cycles to process a packet. Queuing delays, which occur when an arriving packet waits in the output buffer as the link is busy with another packet transmission. Transmission delay (for a packet) is related to the transmission rate of an interface, which is the time it takes the interface to push the packet to the wire.

Transmission speed part 1:
To calculate the RTT, we first calculate the total distance and the total mb transferred per second:
Total distance: $20 + 5 + 750 = 775$m
Total speed: $54 + 100 + 2 + 1000 = 1154$mb/s
We then calculate the queuing delay and subtract it from our calculated RTT, finally multiplying our answer by 2.
Queuing delay: $2 + 1 + 5 + 24 =$32ms or (0.032s)
RTT $= (\frac{775}{1154}) - 0.032 = 0.64$
$0.64 \cdot 2 = 1.28$ seconds
We have not factored in the propogation delay, as the value would have little to no effect on the RTT, and is therefore negligible.

Transmission speed part 2:
640KB = 0.64mb
Since we are given the upstream connection speed in the question, we divide the total data by the speed:
$\frac{0.64}{2} = 0.32$s
We take the queuing delay into account and get the following answer:
$0.32 - 0.032 = 0.288$s

2.2 HTTP

HTTP semantics part 1: First of all, it is important to mention that `GET` and `POST` are two different forms of HTTP request. The way they are different is the way form data is sent to the server. The purpose in the method field is to sent data to the server. The `GET` is including all the required data from the server. POST, on the other hand, submits the data to be processed. Another difference between them is especially regarding safety. Here `GET` will be the easiest to hack because crawlers or robots have the ability to use it arbitrarily, which will allow them to manipulate any side effects that `GET` has made.

HTTP semantics part 2: The `host header` is important because it makes is easier to have more than one host. This is because the host head tells the server which of the virtual hosts to use. It is therefore necessary to have a host header if you want to avoid errors by multiple users. If there are multiple users online at once, and there is no host header to tell the server which user is using the various features, there may be problems in the output.

HTTP headers and fingerprinting part 1: The purpose of the set-cookie header is as the header mostly states is to set the cookie. Much of the syntax of this header states when it should expire, where it should be sent and other crucial info. The other header is called cookie and relates nicely with the other header as this header shows the cookies sent by a given server with the use of set-cookie

HTTP headers and fingerprinting part 2:

2.3 Domain Name System

DNS provisions: The DNS is not centralized that way avoiding having a single point of failure, high traffic volume, the database being too distant and constantly having to be under maintenance. Having a centralized database doesn't scale well and will buckle under the excessive growth of the internet. The problem of scalsbility is avoided by DNS being distributed among different classes of server in a hierarchical fashion. These servers are named root DNS servers, TLD DNS servers and authoritative DNS servers. These servers heighten their efficiency by being able to use caching and saving the ip addresses of the most used domains so the lookup process is cut short and doesn't need to go through the whole hierarchical tree.

DNS lookup and format part 1: CNAME helps when an IP address needs to have more than one domain name linked to the ip adress. The advantage of using CNAME is if the ip address of the main domain is changed then all entries pointing to that domain will also be changed. This is because the other domains simply redirect to the main domain instead of using the same ip address. DNS load balancing works by sending ip adresses in different orders each times so different clients use different servers. This works because clients usually use the first ip address sent by the DNS server.

DNS lookup and format part 2: he main difference between iterative and recursive lookups is that in the iterative lookup most of the burden is placed on the local DNS server whilst with recursive lookup it is placed on the other servers.

Iterative lookup works by the host contacting the local DNS server with a request that request is then passed along to the root Server then returns a list of TLD DNS servers to the local DNS server. The local DNS server then contacts a TLD DNS server which then returns a list of authoritative DNS servers to the local DNS. The local DNS server then contacts a authoritative DNS server which finally returns the ip address to the DNS server which then passes that along to then host. The main key thing about this method is that the local DNS server is very central when it comes to requests. It is the main center of traffic that is returned to often and is burdened until the actual ip address is received.

The recursive method works by not returning to the local DNS as often. The lookup first from the host then to the local DNS server, then to the root DNS server which then request from TLD DNS server which request from authoritative DNS server which has the ip address. Then it just sends it back the way it came in a reverse order all the way back to the host.

Recursive lookups are justified when used with caching. The initial lookup to the local DNS server is in most places recursive since it requests another server to do the lookup on its behalf. And from that point on if the ip address that is needed is used by other people often then it is most likely cached and the other servers can be bypassed and the needed ip can be retrieved quickly.

3 Design and Implementation

3.1 Protocol

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 %)

We have designed our protocol to be independent of any character strings or character combinations.

Although there is a hardcoded “start request" string that is passed to the server on each request (login, lookup and logout), there is no corresponding “end request" string.

The first request from the client must be a “request" string, one of (LOGIN, LOGOUT, LOOKUP). The subsequent messages will depend on the request type. So, in the case of a LOGIN request, the subsequent requests must be the username, password, port and ip in that order. Finally, each message must be followed by a newline character. This is perhaps not a very efficient design decision since it requires not only unnecessarily many reads/writes to the file descriptor but also makes the server-side code messy and complex (discussed below).

However, this also means that there are no other restrictions on the client side. That is, the username and password can for example be any character since no characters are reserved for transmission between the client and server. This is the beauty of not having any dependencies in the protocol. It offers huge flexibity in terms of user input. In fact, the username and password can both be "LOGIN" and the program still works as expected.

In the server we use the length of the arriving request and the number of requests expected, to determine when a particular request starts and terminates. The ip and port have fixed lengths so this is easily implemented for them, but for user generated input like username and password, we determine the length of the input using a loop and the fact that that strings are null terminated in C.

3.2 Technical implementation

Peer

Document technical implementation you made for the peer - cover in short each of the six TODO's for peer.c. (Approx. weight: 15 %)

We use the Open_clientfd helper function to set up a connection to the name server. The Rio_readinitb function is used to associate read buffer to the server socket.

The Rio_writen helper functions are used to write to the server socket. The Rio_readlineb function is used to receive responses from the server. These are used as part of while loops.

The login request has already briefly been mentioned above. It is simply a series of writes to the server socket, where each request is separated by a newline character. Before setting logged_in to 1, we use the Rio_readlineb function to confirm a success response from the server.

In the case of the lookup command, where the expected number of responses (number of logged in users) cannot possibly be known by the client, we use atoi() to convert the first response (holding the number of users) to an int variable. This acts as a counter so that the client can know how many responses are expected and thus when the loop should terminate.

For logout, we send the username along with the LOGOUT request, so that the server can know which user to logout. Furthermore, if the user attempts to exit before logging out we ask him/her to logout first. It could also be possible to handle this on the server and make a logout followed by an exit request on behalf of the user, but it would be overly complex to implement.

We remember to close the connection the server socket on exit. The control flow of the client side code, ensures that the user is properly logged out before the connection is closed.

Server

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 %)

We used a multithreaded model to handle multiple clients. This obviously required a mutex since a global shared variable (client struct) is accessed by clients. The Rio_readlineb and Rio_writen are used for read and write operations. They use a buffer and thus offer great stability when reading/writing to/from connected file descriptors.

Since there is no dedicated “end" string to indicate when a particular request has finished sending all its sub-requests, we use an enum and multiple flags in the server to determine the length of a request.

The first request is always one the request types (LOGIN, LOGOUT, LOOKUP) and so we use an enum and set its value to one of these whenever they are received. The value of this enum is then changed when a particular request is finished (and all its sub-requests have been received). As mentioned above, since C strings are null terminated, we use a while loop to count the length of user generated input (such as username and password). We use either the number of requests (for the lookup request) or the order of requests (for login) to determine when a particular request has finished transmitting all its sub-requests. In the case of login for example, we use the fact that the ip is sent last and that it has a fixed length to assert that no more requests of type LOGIN will be received next. Thus, we change the enum value after ip has been received to indicate that LOGIN requests are finished. On the other hand, in case of a lookup command, the enum is simply changed when the client struct has been searched through (in a loop) and a response has been sent to the client.

Misc.

We use strdup() to store a copy of the username on the client-side to be used later in the logout request. This requires a subsequent free() which is called on exit.

Since we don’t explicitly reap threads, we call Pthread_detach and free in the worker threads.

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%)

For large or important applications it is not a very good idea. A centralized server has a single point of failure, on the other hand, it is easier to manage and troubleshoot if/when issues arise. However, centralized servers are harder to scale if load on the server increases. The server can be changed to a distributed server by using additional hosts to act as servers and directing client requests randomly to one of the servers or alternatively, to the one with least load.

4 Testing and Shortcomings

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 %)

We have tested the implementation in different ways to make sure that it works correctly and does not give unprecedented errors. In terms of testing, we have attempted to perform negative testing to ensure that the program returns errors when a specific input is not recognised or incorrect. If any unrecognised client username or password is entered, the server returns an "error" statement, it does the same for any incorrect number of arguments entered. We also tested the code by entering LOGIN and LOOKUP as username and passwords, considering that we used LOGIN and LOOKUP as requests. We expected that this might confuse the system or cause errors, but it did not. The code works if the client username and password is set to any letter, word, or mathematical symbol.

One of the significant bugs in the code occurs when a login is unsuccessful, and the user attempts to login a second time into the same server. This results in an error, even if the username and password is correct the second time. However, the implementation works as intended as soon as a new server is created. These bugs are due to the complexities in the server code which is discussed in a section above. *Usernames and passwords are hardcoded in the DBSETUP() function in name_server.c These can be used when testing the program.*

5 Conclusion

The programming section of the assignment was implemented as intended, such that the chat service using client-server architectures works accurately. We have also tested our implementation so that it is robust and we can ensure that it works as intended. Lastly, we have discussed any shortcomings and bugs we have encountered.