CS 433 : Computer Networks (Project)
Design, Architecture and Testing for Mini-LinkedIn

Indian Institute of Technology
Gandhinagar

Submitted By

Aditya Tripathi 18110010
Kushagra Sharma 18110091
Nishikant Parmar 18110108

Table of Contents -

High Level Design (HLD)

General Description

Mini-LinkedIn will consist of several different features, taking inspiration from the popular social networking platform LinkedIn. The server and client model for Mini-LinkedIn works on TCP protocol, where the server serves the requests by the client(s). The server interacts with the database to retrieve, insert, delete and update data of the users/companies. The client-server-database architecture is shown in Fig. 1 below.

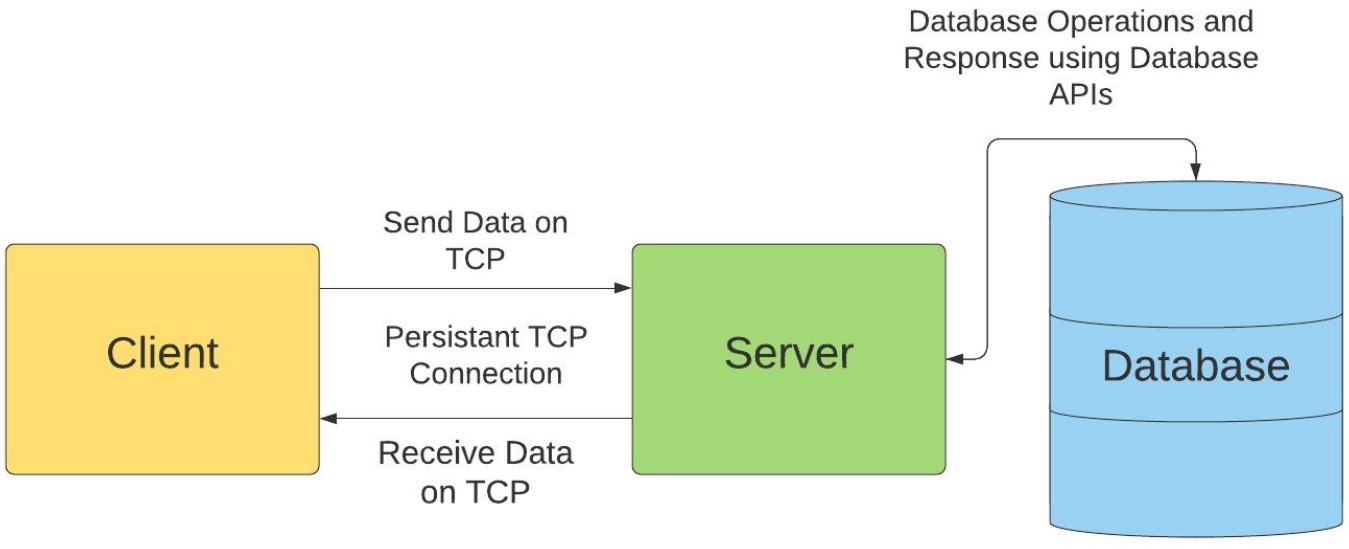


Fig. 1

The server broadcasts its available states (routes) to the client and serves the requests by transitioning into appropriate states by receiving and/or sending data to the client. The server should maintain session information regarding each client. The server acts as a finite state machine(FSM) as shown in Fig. 2 below.

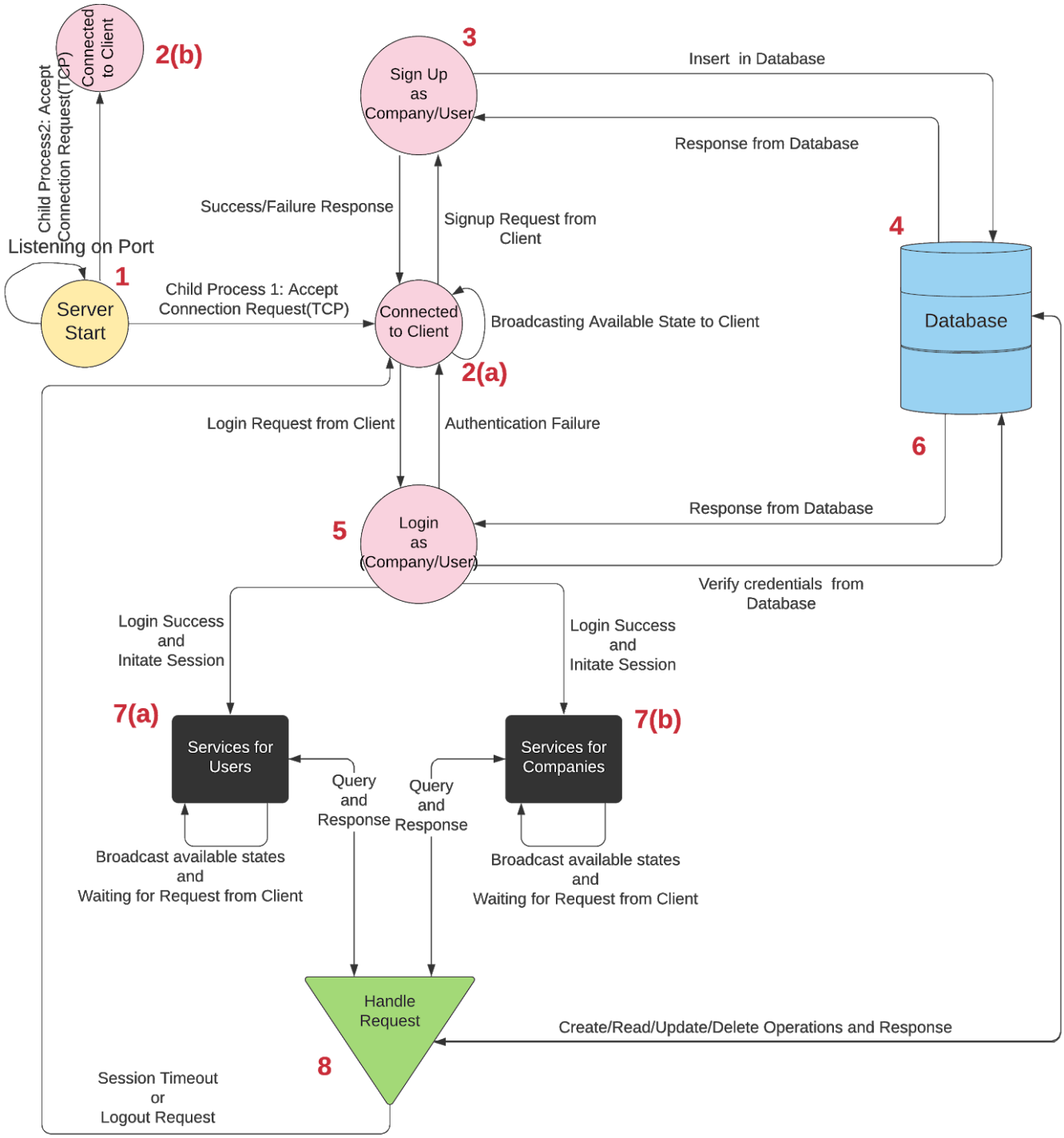


Fig. 2

The client should be able to use Mini-LinkedIn without understanding the low-level functioning of the backend. Privileged features like updating profile, getting feed etc. should not be accessible by the client without login. After logging in, the session should be available for a limited duration and the client should be logged out if the session expires or if the client chooses logout. The client architecture is shown in Fig. 3 below.

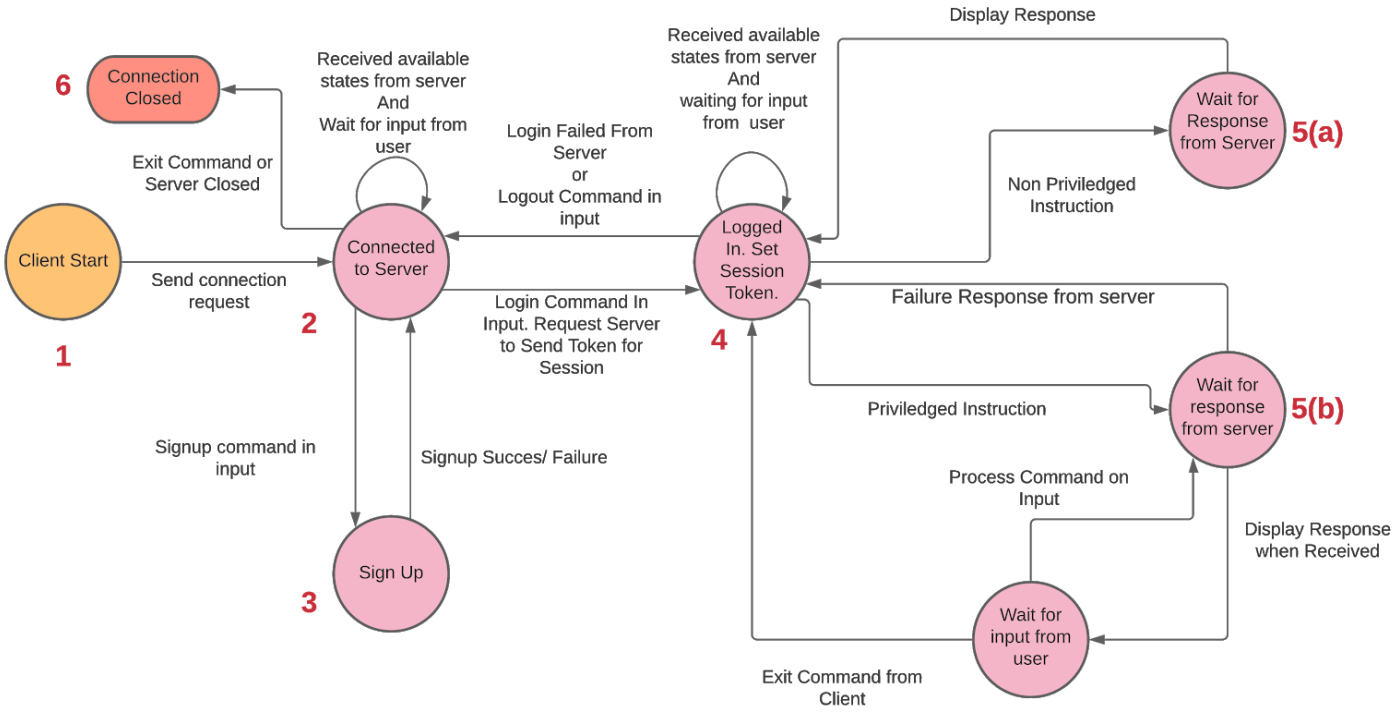


Fig. 3

Note - Non-privileged instructions are those where the client enters a command and receives response from the server whereas **Privileged** instructions are those where the first client enters a command and waits for response from the server. Now based on the response from the server the client is **asked again** to input his action under that command.

For instance, when a client asks for applying to a job (first input provided by client) then server responds with all the jobs then client enters which job to apply (second input provided by client under applying to jobs command). These are shown in the bottom right part in the above diagram.

Also, **logged In and out** states are different from **Privileged** and **Non-Privileged** instruction states and **not to be** confused or used interchangeably as clarified above.

Networking Paradigm

Client and Server Interaction -

Since, TCP is a reliable data transfer protocol, all the transactions between client and server happens on TCP. The server opens up a socket and listens, as soon as a client sends a connection request, a TCP connection is set up between the client and the server and is persistent for the entire communication between them until any one of them closes the connection.

These are the steps in data exchange -

1) Data is encoded in a proper format by the sender which both the entities can understand.
2) It is payloaded directly in a TCP packet and sent to the receiver.
3) The receiver decodes data and takes appropriate actions.

Here, sender/receiver can be both client and server.

Hence, both the entities exchange data with TCP as the underlying protocol.

Server and Database Interaction-

The server communicates and performs operations on the database directly by using APIs provided by the database (happens over HTTP).

Feature Checklist/ Addressing Proposal

The Mini-Linkedin proposal can be found here. As specified in the proposal, we have implemented all the **basic features, advanced features, security features, and bonus features (grace points)** except adding GUI.

There are many use cases handled by the Mini-Linkedin. We can list them here:

- Different login and signup flow for a user and a company.
- Users can post and react to their connection's posts.
- Users can send connection requests to other users on Mini-Linkedin.
- Users can edit and create their profile where they can add skills, experience etc.
- Users can accept connection requests from other users.
- Users can view the people who have viewed their profile.
- Users can endorse skills of their connections. And users can also see the people who have endorsed their skills.
- A company will be able to post jobs, and view the profile of users who have applied to the job.
- Users can search for jobs based on their skills as filters. The search query will be made on all the available jobs on the Mini-LinkedIn platform.
- The server is scalable and can handle concurrent client requests.
- The password entering field at login/signup is masked to hide it while entering.
- There are multiple reactions available for posts - like, support and clap.
- Users can comment on any posts in their feed
- Session management is also done. At the time of login, a session for the user will be created, and after a certain amount of time, set in the session limit, the user will automatically be logged out of the Mini-LinkedIn system. The user will have to re-login to continue.

Low Level Design (LLD)

Implementation:

Client-Server - Node.js

Node.js is an open-source, cross platform runtime environment for executing JavaScript code outside of a browser. It works on an event based model requiring callback functions which provides better performance than traditional thread based models.

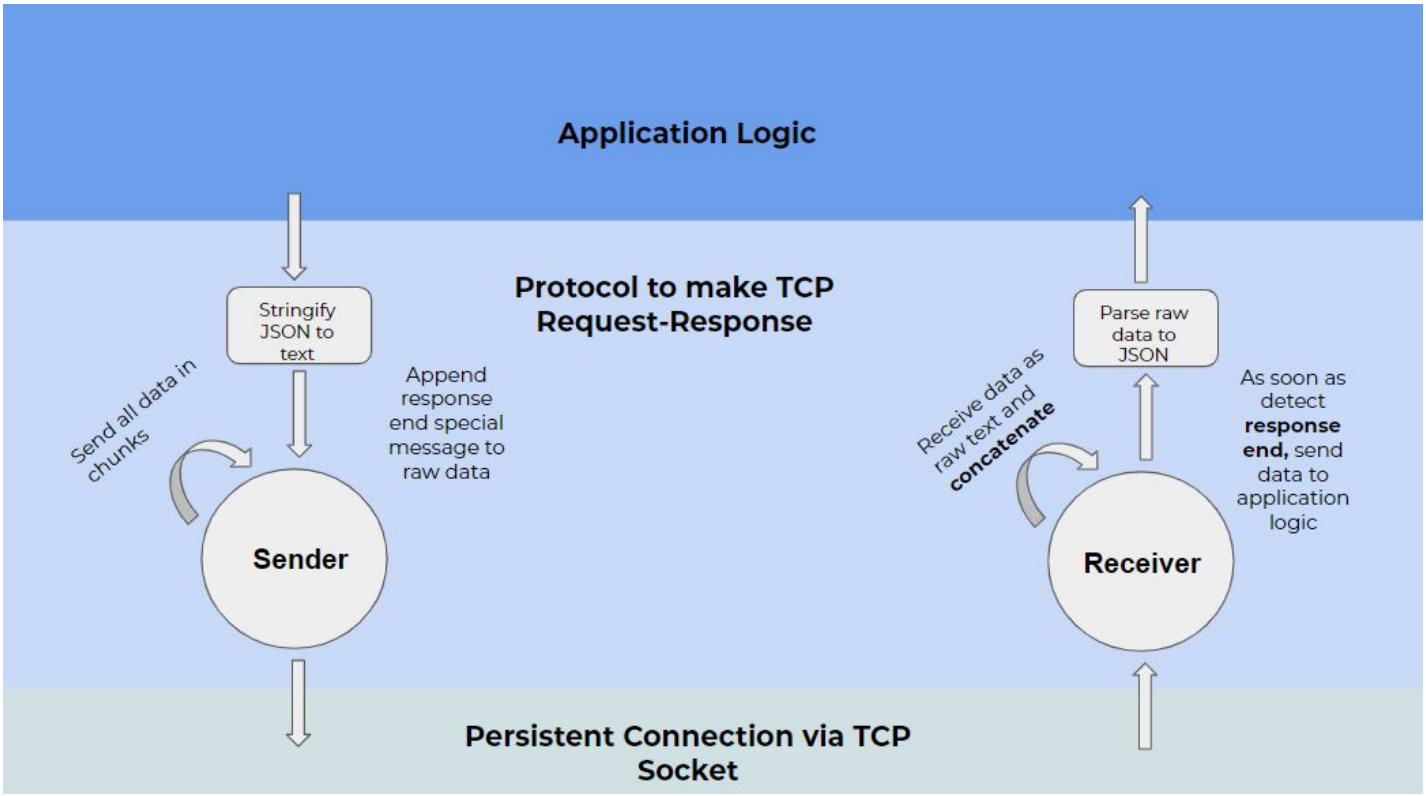
A TCP connection can be easily set up on Node.js using sockets that allows client-server interaction in Mini-LinkedIn.

Database - MongoDB

MongoDB is a document-based NoSQL database with powerful querying capabilities, flexible schemas which suit the requirements of the Mini-LinkedIn tool. It is nicely compatible with Node.js, offers code-native data access (APIs) for the server to work and has enough documentation.

Protocol over TCP, Encoding and Decoding data in TCP packets

Data between the client and the server are sent by packing data into JSON format. After packing the data is sent via the TCP sockets in the server and the client. Both the client and the server parse the received data to perform the respective actions.



For sending one single request or response -

The entire JSON data (be it sending a request from client for some command or sending response from the server for a command from client) is first made into a string by stringifying the JSON. Then this string is converted into a **raw string of hex**. Depending on the **buffer size** defined on the client and the server side, we create chunks of the string.

For each such complete raw string (formed from JSON) an **end of response is marked**

using special characters (***EOI*** *(End of Instruction)* in our case). Now, these packets are sent over TCP socket as a stream of bytes and are received by the other side socket. That socket keeps on collecting the received data, and checking at each stage if the **EOI** is received.

For receiving one single request or response -

The receiver receives raw text in a **loop** from the other party and tries to concatenate all the raw text received. For each text received, it also checks whether the current text is the end of this whole command's response/request with the special end marked using special characters. When it detects the end of the message it sends the entire raw text string (that was formed by concatenating many packets) to the **application logic** where it can be parsed into JSON and processed and break current loop and go on for **receiving next packets.**

This is how the sender and receiver knows where a particular data ends which makes a transaction between client-server as request-response message.

Note: There is no extra overhead for sending JSON over TCP, as we stringify the JSON before sending it as a payload. Therefore, we are effectively sending a string over TCP and parsing that string at the other end of the connection.

The JSON structure is as follows -

Client to Server -

```
{
  'command' : <type:String, instruction/route for server>,
  'body' : <type:JSON, data for server>,
  'token' : <type:String, client's token for session management>
}
```

Server to Client -

```
{
  'status': <type:String, status code of request handled by server>,
```

```
  'message' : <type:String, message corresponding to the status code>,
  'data' : <type:JSON, payload after handling request>
}
```

Server Side Session Management

When client requests for logging into the platform the server makes a JSON token (a string) by encoding the following client details -

1) Primary key/ unique ID of the user/company from the database
2) A flag to detect whether a client has logged in as a user or as a company.

Then it sends this token to the client for session management in the response of login success.

Client Side Session Management

After a login attempt, the client gets a response from the server. If this response is a success, there will be a token in the received response. From this point, the client sets the token value in its header data, the data is sent as a query to the server. This token is always present in the token field of the client to server JSON. The case where the token in the header is unset is if the user logs out, or its session expires.

This token is used to verify the login state of the user on the server side. This check is performed each and every time the user makes a request.

Authentication, Commands and Action Management

The client chooses one of the available states broadcasted by the server and sends the respective route using the `command` field in the JSON object from the client to the server.
If the command field contains `signUp` or `login`commands the server queries the database and handles them. In case of `login` a JSON token is generated which is sent back to the client.

For all other commands the server checks after decoding the token and verifying fields of the client for its expiry and authenticity. The data required for handling a command is

sent in the ‘body’ field of the JSON object and parsed at the server side to handle the request. If the command is handled without any errors, a corresponding message of ‘success’, status code ‘200’ and respective payload is sent to the client. Otherwise, the respective error code and message with no payload is sent to the client.

Security aspects

- As mentioned above, for every coming request from the client, the server verifies the token and command and then takes necessary action. If the authentication via the token fails the commands will not be handled by the server unless the client logins with proper credentials.

- The client is not permitted to perform any command except login/sign up without logging in.

- A user cannot access/use APIs for a company and a company cannot access/use the APIs for a user. For instance, only a user will be able to send connection requests to other users and a company cannot send connection requests. Similarly, a user will not be able to post jobs, only a company will be able to post jobs.

- The passwords entered by the client are obscured using (*) symbol for security.

- All the data sent over TCP connection is encoded before sending.

- A user can only see posts of users who are in their connection. Similarly, a user who is already a connection of another user, can’t send another connection request. Similarly, a liked post cannot be liked again by the same user.

Commands/APIs Description:

The expected data format is mentioned **below each command**. All this data is inserted in the **body** field of JSON mentioned above from client to server.

Normal Commands

- **login:**
  A. Users will be able to login to the Mini-linkedIn system. After login the user will enter a logged in state. In the logged in state, the server will be able to provide more states/API's to the client to work with. For instance, the user will now be able to send connection requests to other users, apply to jobs posted by companies, post and like/clap/support other users posts, etc.
  B. Companies will also be able to login to the Mini-MinkedIn system using the same command. After login, the company will enter a logged in state. In the logged in state, the server will be able to provide more states/API's to the company to work with. For instance, the company will now be able to create jobs where other users can apply, view the profiles of the users who have applied to those jobs, etc.
  The server distinguishes each login into a user or company by finding their credentials stored in the DB and sets whether the client is a user or company in the token information.

  { "email" : " ", "password" : " " }

- **signUpUser:** Using signUpUser, the user will be able create an account in the Mini-linkedIn system.

  { "firstName" : " ", "lastName" : " ","email" : " ", "password" : " " }

- **signUpCompany:** Using signUpCompany, the company will be able create an account in the Mini-linkedIn system.

  {"companyName" : " ", "description" : " ", "address" : " ", "email", : " " "password" : " "}

- **logout:** Using logout, a user or a company will be able to give up their logged in state, and their sessions will be cleared. They will enter in a state where they will either be able to send a login request or a signup request again.

  { }

- **getMyProfile:** Using getMyProfile, a user or a company will be able to fetch and view their own respective profiles.

  { }

- **updateProfileUser:** Using updateProfileUser, a user will be able to update its profile's field, as defined in the User Schema.

{”status” : “ ”, ”title” : “ ”, ”address” : “ ”, ”skills” : “ ”}

- **deleteAccount:** Using deleteAccount, a company or a user will be able to delete its account from the Mini-LinkedIn system.

  { }

- **getMyFeed:** Using getMyFeed, a user will be able to get its feed. The feed of a user will display all the posts by the connections of the user or the user itself. A user will be able to react on the post. A user will not be able to view and react to the posts of users who are not in their connections.

  { }

- **postJob:** Using postJob a company will be able to create a job. This job can be searched by other users, and can be applied by users.

  {
  “jobType”: “” ,” jobLocation”: “” ,”description”: “” , “employmentType”: “”
  ,”industryType” : “”, “experience”: “” , ”budget” : “” , ”skillSet” : “”
  }

- **createPost:** Using createPost, a user will be able to create a post, which can be seen by the connections of this user on their feed.

  {“content”: “”}

- **searchJob:** Using searchJob, a user will be able to search all available jobs on Mini-LinkedIn with filters on its skill sets. He can then apply for these searched jobs, using the applyToJob command.

  {“skills”: “”}

- **feedCompany:** Using feedCompany, a company will be able to see all the jobs posted by the company. A company will only be able to see the jobs posted by it, and not of any other company.

  { }

Privileged Commands

Privileged commands are nothing but 2 step commands, i.e they require 2 levels of input from the user. The first level is when a basic command is sent(take **commentOnPost** for example). The body sent for a privileged command contains a field called **index**. This index is set to **-1** when the first level call is made. Now the client goes into a different state where it is expecting some data from the server. On the server side, the server checks the value of the index. If the index is **negative,** then the server sends the complete list of information depending on command passed (in our example case, the server will send an array of all the posts in the feed of the user).

Now comes the **2nd level** of input from the client. The client sees the array and enters the which of the values it wants to interact with (in our example, it will give the number of the post it wants to comment on, say 10 for example and also enter the comment text which he wants to comment). Now after entering these, the client will send the data to the server, with the same command name, but the index value will be set to some **non-negative integer.** The server will check that the value if the `**index**` field in body is not negative. Therefore, the server will change its state and access other details (such as post_id and comment_text in our example along with the index field) provided by the client, and will perform the appropriate task.

The client will not leave its privileged state, and will keep on asking for a number of post which the client wants to comment on. If the user wants to exit the privileged state and enter the normal state, then it should enter `**exit**` to change the state back to non-privileged state, where it can now access all the given commands as usual.

Hence, in privileged instructions the client goes in a tight loop (2 level deep).

- **like/clap/support :** Using like/clap/support, a user will be able to react to a post in their feed. A single user can like, clap and support a post only once. An already reacted message will be displayed on further interaction with like/clap/support on the same post.

```
{       “index” :
        (-1 - for server to send all the posts in feed,
        1 - for server to like/clap/support a post with id mentioned) ,
        “id” : “”
}
```

- **acceptConnection:** Using acceptConnection, a user will be able to accept a connection request from the available list of users who have sent him/her a connection request earlier. A connection request can only be accepted once, after that the user1 will be added as a connection of user2, and user2 will also be added into user1's connection list.

  { “index” :
  (-1 - for server to send all pending connections of current user,
  1 - for server to accept connection of a user given by id) ,
  “id” : “”
  }

- **sendConnection:** Using sendConnection, a user will be able to send a connection request to any person on the Mini-LinkedIn system, provided that user is not a connection of the current user. User1 will be added in the connection request queue of the user2 and user2 will be added to the sent connection queue of user1. A user will be able to send a connection request only once. Upon further requests, the current user will be displayed with a message stating “request already sent”.

  { “index” :
  (-1 - for server to send all users that can be connected,
  1 - for server to send connection request to a user specified by id) ,
  “id” : “”
  }

- **endorseSkill:** A user1 can endorse skills of a user2 on the Mini-LinkedIn system. The user2 must be a connection of user1. The endorsement is per-skill, therefore, upon endorsement of a skill of user2, user1 will be added to the endorsement list of user2's perspective skill. Endorsement of a specific skill by a specific user can only be done once by a single user.

  { “index” :
  (-1 - for server to send all the users in connection,
  1 - for server endorse skill_indexth skill of user specified by user_id) ,
  “user_id” : “”,
  “skill_index” : “”
  }

- **applyToJob:** Using applyToJob, a user can see all the job postings on Mini-LinkedIn, and based on that, it can apply on any job. It can only apply to the selected job, if the job is open and if it had not previously applied to the same job.

```
{
        “index” :
        (-1 - for server to send all jobs present in Mini-Linkedin,
        1 - for server to apply to a job specified by id field) ,
        “id” : “”
}
```

- **viewProfileUser:** A user will be able to view the profile of any user on the Mini-LinkedIn system. If it is the first time the user1 has viewed the profile of user2, then user1 will be added to the ‘viewedby’ array of the user2, and the user2’s profile view count will be increased, and the profile will be fetched by the user1. However, if the user1 has already seen user2’s profile, then, the profile view count of user2 will not change, and the profile will be fetched. A user can't see its own profile using this API, he/she will have to use getMyFeed API.

```
{
        “index” :
        (-1 - for server to only few details of all the users present in Mini-Linkedin,
        1 - for server to send entire profile of a user mentioned by id) ,
        “id” : “”
}
```

- **viewProfileCompany:** Using viewProfileCompany, a company will be able to view its profile. After that, the company can also select a specific job and a specific user who has applied to the job posting. The company can view the profile of that user. If it is the first time the company has viewed the profile of that user, then that company will be added to the ‘viewedby’ array of the user, and the user’s profile view count will increase. However if the company has already viewed the profile of the user earlier, then the profile view count will not increase.

```
{
        “index” :
        (-1 - for server to send entire profile of the company,
        1 - for server to send entire profile of user given by id) ,
        “id” : “”
}
```

- **connectionRecommendation:** With this command, a user can get some connection recommendations to which he may want to connect with. First, user is shown all the connection recommendations and then he can choose which one

he wants to connect to. A user is recommended iff it is in the connection of any of the connections of the current user and is not in direct connection with the current user.

```
{
        “index” :
        (-1 - for server to send all users that can be potential recommendations for connection,
        1 - for server to send connection request to a user specified by id) ,
        “id” : “”
}
```

- **jobRecommendation:** Using jobRecommendation command, a user will get job recommendations then he can choose which one to apply to. A job is recommended to the user iff the intersection of skills needed in the job and skills of the user is not null.

```
{
        “index” :
        (-1 - for server to send all the recommended jobs to the user,
        1 - for server to apply to a job specified by id) ,
        “id” : “”
}
```

- **commentOnPost:** With this command a user will be able to comment on any post. Initially the user will be able to see all the posts in his feed then can choose any post and write a comment on that post. A user can write any number of comments on any post provided that the post belongs to the user’s feed.

```
{
        “index” :
        (-1 - for server to send all the posts in the feed of the user,
        1 - for server to comment comment_text on a post given by post_id ) ,
        “post_id” : “”,
        “comment_text”: “ ”
}
```

Codebase Directory Architecture:

Mini_LinkedIn

```
├── README.md
├── client
│   ├── client.js
│   ├── commands.js
│   ├── package-lock.json
│   └── package.json
├── mininet
│   ├── func_def.py
│   ├── gen_data_company.py
│   ├── gen_data_user.py
│   ├── mini_test_company.py
│   ├── mini_test_user.py
│   ├── test_company.py
│   ├── test_user.py
│   ├── workload_company
│   │   ├── address.txt
│   │   ├── budget.txt
│   │   ├── companyName.txt
│   │   ├── description.txt
│   │   ├── emails.txt
│   │   ├── employmentType.txt
│   │   ├── experience.txt
│   │   ├── index.txt
│   │   ├── indexOfApplicant.txt
│   │   ├── indexOfJob.txt
│   │   ├── industryType.txt
│   │   ├── jobLocation.txt
│   │   ├── jobTitle.txt
│   │   ├── jobType.txt
│   │   ├── password.txt
│   │   └── skills.txt
│   └── workload_user
│       ├── address.txt
│       ├── content.txt
│       ├── emails.txt
│       ├── fnames.txt
│       ├── index.txt
│       ├── indexOfJobs.txt
│       ├── indexOfUser.txt
│       ├── lnames.txt
│       ├── password.txt
│       ├── skills.txt
│       ├── status.txt
│       ├── title.txt
```

```
│   └── viewUserIndex.txt
└── server
    ├── config.json
    ├── db
    │   ├── constants.js
    │   └── db.js
    ├── errorHandler.js
    ├── feed
    │   ├── companyFeed.js
    │   └── userFeed.js
    ├── models
    │   ├── company.js
    │   ├── jobPosting.js
    │   ├── post.js
    │   └── user.js
    ├── modules
    │   ├── databaseAPI.js
    │   ├── helper.js
    │   ├── jobs.js
    │   ├── posts.js
    │   ├── serverAPI.js
    │   └── userControl.js
    ├── package-lock.json
    ├── package.json
    └── server.js
```

Database Schemas

Apart from these fields a unique id / primary key is generated as soon as a record is inserted which is used for determining a record later for other operations.

User Profile -

```
    firstName : {
        type : String,
        required : true
    },
    lastName : {
        type : String,
        required : true
    },
    password :{
        type : String,
```

```
        required : true
    },
    status : {
        type:String
    },
    title : {
        type:String
    },
    email : {
        type : String,
        required : true
    },
    address : {
        type : String,
    },
    skills : [
        {
            skillName : {
                type:String,
            },
            endorsedBy: [
                {
                    type:String
                }
            ]
        }
    ],
    profileImg : {
        type : String
    },
    connections:[
        {
            type:String
        }
    ],

    // Sent not accepted
    connectionRequestsSent:[
        {
            type: String
        }
    ],
```

```
    // Received but not accepted by this user
    connectionRequestsReceived:[
        {
            type: String
        }
    ],

    experience : [
        {
            title:{
                type: String,
                required: true
            },
            companyName:{
                type: String,
                required: true
            },
            description:{
                type:String
            },
            startTimeStamp:{
                type: Date,
                required: true
            },
            endTimeStamp:{
                type: Date
            }
        }
    ],
    education : [
         {
            instituteName:{
                type: String,
                required: true
            },
            description:{
                type:String
            },
            startTimeStamp:{
                type: Date,
                required: true
            },
            endTimeStamp:{
```

```
            type: Date
        }
     }
],
appliedToJobs: [
    {
        companyId: {
            type:String,
            required: true
        },
        appliedAt:{
            type: Date,
            required: true
        }
    }
],
viewedBy: [
    {
        id: {
            type : String
        },
        isCompany : {
            type: Boolean
        }
    }
],
posts: [
    {
        postId: {
            type: String,
            required: true,
        },
        postedAt: {
            type: Date,
            required: true
        }
    }
]
```

Company Profile -

```
companyName : {
        type : String,
        required : true
    },
    password :{
        type : String,
        required : true
    },
    email : {
        type : String,
        required : true
    },
    description: {
        type: String,
        required: true
    },
    address : {
        type : String,
    },
    profileImg : {
        type : String
    },
    jobsPosted: [
        {
            type: String
        }
    ]
```

Post -

```
    postById : {
        type : String,
        required : true
    },
    content : {
        type: String,
        required : true
    },
    images:[
        {
```

```
            type : String
        }
    ],
    likes: [
        {
            type: String
        }
    ],
    claps: [
        {
            type: String
        }
    ],
    supports: [
        {
            type: String
        }
    ],
    comments:[
        {
            user_id:{
                type:String
            },
            comment_text:{
                type:String
            }
        }
    ]
```

Job Posting -

```
    companyName : {
        type : String,
        required: true
    },
    companyId : {
        type: String,
        required: true
    },
    companyLogo : {
        type : String,
```

```
        required: false
    },
    jobTitle : {
        type : String,
        required: true
    },
    jobType : {
        type : String,
        required: true
    },
    jobLocation : {
        type : String,
        required: true
    },
    numberofApplicants : {
        type : Number,
        required: false,
        default: 0
    },
    description :{
        type : String,
        required: true
    },
    employmentType : {
        type : String,
        required: true
    },
    industryType : {
        type : String,
        required: true
    },
    experience : {
        type : Number,
        required: true
    },
    budget : {
        type : String,
        required: false,
        default: "Not Disclosed"
    },
    isActive:{
        type : Boolean,
        default : true,
```

```
    },
    skillSet: [
        {
            type: String,
            required: true
        }
    ],
    applicants: [
        {
            userId: {
                type:String,
                required:true
            },
            appliedAt:{
                type: Date,
                required: true
            }
        }
    ]
```

Testing

Note: We are using a local instance of MongoDB for testing. And the connection string for the same has been configured in the `/server/db/constants.js`. If you want to run the code, kindly check the instructions for setting up the database for our testing from the github repo or in this section of the document.

Manual Testing Plan

To perform manual testing we created an interactive client side logic, which interacts with the user through the terminal window. Though we are sending a stream of bytes in a pure TCP connection between client and server, we used the logic described above to demarcate between different commands. And using minimal states in the client side, we were able to achieve the interactive client logic.

- We had to define the input structure of all the commands that were possible on the Mini-LinkedIn. We did that in the file commands.js(as can be seen in the

directory structure). After that, using the `prompt` module in Node-JS we performed I/O operations via STDIN and STDOUT.

- When a client first connects to the server, the server lists all the available commands that the client can perform, which is in accordance with the state of the server. The client can then send appropriate commands, along with any input data that is necessary.

- These state machines keep on interacting with each other until the client says to terminate the connection. After termination the TCP socket is closed.

Here is the list of all the available commands that the client can access.

```
const commandsArray = {
    '0':'exitProgram',
    '1':'login',
    '2':'signUpUser',
    '3':'signUpCompany',
    '4':'logout',
    '5':'getMyProfile',
    '6':'updateProfileUser',
    '7':'deleteAccount',
    '8':'getMyFeed',
    '9':'like',
    '10':'clap',
    '11':'support',
    '12':'acceptConnection',
    '13':'sendConnection',
    '14':'postJob',
    '15':'createPost',
    '16':'searchJob',
    '17':'feedCompany',
    '18':'endorseSkill',
    '19':'applyToJob',
    '20':'viewProfileUser',
    '21':'viewProfileCompany',
    '22':'connectionRecommendation',
    '23':'jobRecommendation',
    '24':'commentOnPost'
```

```
}
```

Code Dependencies

- nodejs
    - bcrypt: "^5.0.0"
    - bcryptjs: "^2.4.3"
    - express-jwt: "^6.0.0"
    - mongoose: "^5.10.11"
    - mongoose-timestamp: "^0.6.0"
    - jsonwebtoken: "^8.5.1"
    - prompt: "^1.0.0"
    - util: "^0.12.3"
    - prompt-async: "^0.9.9"
- python
    - pickle
    - faker
- mongodb

Note: All the node js dependencies can be installed automatically, by running `npm install` in the corresponding directory.

Running Manual Testing

- Make sure that a local instance of mongodb is running. Fire up the terminal and type `mongo`. Then type `use linkedin`. Then type `exit`.

- Kindly configure the IP of your system in the /server/db/constants.js.

- Go to `/server ` and execute command `npm install`to install nodejs dependencies.

- Use the command `node server.js localhost` to start the server from the directory `/server`

- Go to `/client` and execute command `npm install` to install nodejs dependencies.

- Use the command `node client.js localhost` to start the server from the directory `/client/`. You are good to go. Follow the instructions on the terminal window.

Automated Testing Plan

We divided the testing in 3 parts. The first was generating a random workload for the automated testing. The second was creating scripts that can run these tests in an automated fashion. Third was to create mininet topologies and run the client and the server on the mininet nodes, with horizontal scaling.

Part- I: Workload Generation

- Generated random workloads using `pickle` and `faker` and `random` modules of python for user and company. The workloads consisted of all the fields mentioned in the user and company schema as well as data for job postings and creating posts. Separate files were created for each field of the user and company schema.

- An index file was created with random numbers as input and used this index file to pass the specific entry of the workload to the client. The data was generated for 100 users, i.e. the workloads contain credentials for 100 users and 100 companies and can be used to test the APIs.

- The index of the user, whose credentials are to be tested can be provided in the automated testing scripts and their credentials can be read from the respective generated files. The read credentials can then be used by the functions defined in `func_def.py` file to run the tests of various APIs provided by the client.

Part- II: Automated Testing of an Interactive client

Creating automated tests from an interactive client was the most challenging task in the testing process. We created a highly **modular testing code,** where we defined functions for all the client APIs with inputs as arguments required by the API to interact with the server. This file can be found in `/mininet/func_def.py`.

- The major part was to ensure that the inputs occur in a blocking fashion i.e if we are testing for the login of a user, then password should be sent to STDIN after we have sent the email, and not before that.

- To ensure a blocking fashion, we used the `pexpect` module of python. Using pexpect we checked what type of input is required by the client. If the client prints 'email' in the terminal, then pexpect (via STDOUT) will know that the user is currently asking for the email, therefore, we can send email via sendline to the STDIN. And the above described scheme has been incorporated for all the API's and their inputs from the generated workload.

- Once we were able to establish a blocking mechanism for the I/O, all we needed to do was to write a file that makes calls to all the methods defined in `\mininet\func_def.py` in a specific manner. This file can be called by different concurrent hosts on mininet nodes.

Part- III: Creating Mininet Topology and Testing

- Using the examples provided in the official documentation of Mininet (docs.mininet.org), we created a topology using the **TreeNet** module provided by Mininet with a **depth of 2 and fanout of 4**. This configuration created a mininet topology with 1 switch having 4 child switches and each child switch having 4 hosts connected to it. Thus the depth is 2 and each node has 4 children attached to it.

- We then added NAT configuration with default values to the topology created above. The NAT configuration was needed because each host has its own IP address and the **database** we created was **running on the VM host machine**, but the server running on a host was unable to connect to a network outside of the network created by Mininet.

- We added the required hosts (**16 for the tests**) and ran the **server on the 1st and 16th host** as they were on the opposite sides of the created topology. Thus, all the clients running in the **left subtree sent their requests to the server running on host 1** while the clients running on the **right subtree, sent their requests to the server running on host 16.** We are using multiple instances of the server, thus achieving **horizontal scaling.**

- We used the **`popen`** command provided in `mininet.utils` to execute commands on the hosts of mininet. The server is being run on 2 of the nodes, and clients are being run on the rest. The client programs are being **run in background**, i.e they all can connect to the server concurrently. The IP addresses of the servers were provided to the clients using command line arguments at the time of process creation.
  Note: `popen` is just an alternative to `cmd`, which helps in execution of command. We choose it over `cmd` but the output shown by popen can be controlled by us and shown in a systematic way.

- The clients connected to one of the two servers and requested as defined in the tests created. The outputs of each client were monitored using `pmonitor` command and printed to stdout. Each client starts by connecting to a server, after which the server responds and the client on the initiation of `prompt:` sends the next request sequentially.

- To check for the printing of `prompt:` to stdout we used `pexpect` to read the standard output of the running client and sent a request only when the server responded. The same testing procedure is used for testing the user and company workloads via the `mini_test_user.py` and `mini_test_company.py` scripts.

Running the automated testing

- First, we will start by generating the workload for the testing. There are two types of workload. One for the company base and one for the user base. The workload can be generated by using the command in the directory `/mininet/`
  User Data: `python3 gen_data_user.py`
  Company Data: `python3 gen_data_company.py`

- Run `pip3 install faker` and `pip3 install pickle` before generating workload

- Now the workload files must have been generated in the directory `/mininet/workload_user` and `/mininet/workload_company`

- To test the server and client, run `sudo python3 mini_test_user.py` for users or `sudo python3 mini_test_company.py` for company.

- The above command creates a mininet topology as specified above and starts the server on the respective hosts.

- To run the clients, several subprocesses are created automatically using the command `python3 test_user.py` or `python3 test_company.py`. These subprocess take the server IPs as input and the index of the user to query and create a client process using the same parameters.

- Then the above scripts run the client APIs in a sequential manner as defined in `test_user.py` and `test_company.py` using the helper functions provided in `func_def.py` file.

- The clients are run concurrently by the script, but the output is designed in such a way that the output is shown in a sequential manner, i.e first for the client running on host 2 then for host 3 and so on.

The Big Picture

Learning Outcomes

- **Core Networking:**

    - Understanding key networking protocol, and articulating the low level data communications and abstractions that allow network clients and servers to communicate across the internet.
    - Thus eventually creating our own protocol stack for communication between client and server.
    - Choosing one protocol (TCP) over another (UDP) and making it persistent.

    - Realizing actual network topology through Mininet Software Defined Networking

- **Using Emulators and Testing:**

    - Using Mininet which is a network emulator helped in creating a network of virtual hosts, switches, controllers, and links.
    - A complete experimental network enabled the scope for development, learning, prototyping, testing and debugging.

- **Teamwork:**

    - Collaborating in virtual mode through call conferencing for hours, working together on shared code live.
    - Clearly identify issues affecting the team's performance, and communicate with each other for seamless working.

- **Communication:**

    - Gathering expectations for the entire project, noting them down and clarifying doubts regarding all aspects.
    - Effectively communicate technical information to audiences in formal, technical documentation and in presentations.

Challenges

- The main crux of the project was to make TCP (which sends data as a stream of bytes) into a request-response transaction for the client and the server to realize the features of Mini-LinkedIn. We faced **challenges** in designing the scheme for data communication between the client and the server. Each single request from client or response from the server had to be made from chunks of bytes to an entire message that can be given to the application logic.

- The **challenge** we faced here was due to the fact that Mininet supported NAT configurations with the help of an **`interfaces`** file stored in **`\etc\networks\`** folder in Ubuntu. But the respective folder and file was obsolete for Ubuntu versions greater than 16.04 and now have been replaced with other

configurations. Since we were running Mininet on VM installed with Ubuntu 20.04 we got a fresh copy of the required file from Ubuntu 14.04 and configured it to function with Mininet.

- After configuring the NAT network, we were required to connect our server to the database. Originally the database was on cloud and we were able to connect to it but even after configuring the NAT network the mininet hosts were not able to contact the cloud database. So we created a local instance of the database on the VM and passed the IP of the VM and the default port of the database running on VM to our server. This configuration enabled us to connect to the database and at the same time provided more authority over the usage of the database..

- The next major **challenge** was to print output of the clients. We used the `pmonitor` command to monitor the outputs of each client. Since `popen` command creates a subprocess and returns it as a popen object, we used a dictionary which mapped the host nodes with the respective popen object and passed it to `pmonitor` to print its output to stdout and the terminate the process after the client has terminated its connection with the server.

- Initially, using the **`cmd`** command all clients were printing simultaneously to stdout and all **outputs** were **interleaved**. To solve the above problem, we used `popen` command and `pmonitor` to print the outputs of each client sequentially although they are running concurrently and requesting the server concurrently.

Future Aspects

- Additional features like posting stories, chatting services, liking and replying on comments etc. can be added to the Mini-LinkedIn client and server feature set.

- A material web or mobile based GUI through HTTP REST can be added.

- Making server APIs public that can be integrated with any front-end device be it native OS app, web app, mobile app or a third-party service.

- From the networking point of view, instead of using a predefined TCP protocol, we can even create our custom TCP/IP stack and try to integrate it with the Mini-Linkedin client-server model e.g. many big organizations are using QUIC

protocol over HTTP such as Facebook etc.

- Designing an automated load balancer that divides the client requests as per the current load on the server instances, and also create and remove server instances as and when needed.