Organización general

https://lucid.app/lucidchart/ca198d66-958b-4556-8b59-2d0e581c762d/edit?viewport_loc=-1086%2C-1908%2C3652%2C1876%2C0_0&invitationId=inv_df3449a3-d385-46f6-b4e3-e1d040998969

Cosas importantes

- El proyecto se debe hacer en un entorno Linux
- Hay que recordar que todo va a estar hecho en C
- Usaremos un switch para la comunicación
- Cuando decimos protocolo nos referimos a la forma en la que será la comunicación entre los proyectos y como el cliente mandará instrucciones o solicitudes al servidor.
- Todos tienen que programar su servidor y su cliente. Pero deben tener en consideración el protocolo porque el cliente del grupo "A" debe poder conectarse al server del grupo "B" y viceversa. Entonces es importante que mandemos la información con el mismo formato. **Véanlo como cuando en web mandábamos solicitudes e información a un API. Solo que en lugar de un json mandamos un texto con un formato específico y también recibimos información con un formato específico.**

Investigación

Enlace recomendado en la entrega:

https://linux.die.net/man/7/ip

Librerías disponibles:

- Mutex
- Sockets
- JSON

Idea de protocolo

De momento tenemos la idea general. Cada uno tendrá su servidor hecho en c en la computadora además de su cliente. Usaremos sockets dentro del servidor para poder manejar la comunicación. A medida que agregamos conexiones al servidor se deben crear sockets que permitan la comunicación entre usuarios.

Puerto para utilizar: 50213

Para trabajar con los elementos, se hará uso de la librería de cJSON para trabajar los datos como objetos de JSON. Los tipos son destacados en negrita y mayúscula. Ese debe ser el nombre exacto que deben usar y que se espera en las respuestas.

El protocolo se vería de la siguiente manera:

1. Registro de usuarios:
    - Cliente a servidor:
      {
        "tipo": "REGISTRO",
        "usuario": "nombre_usuario",
        "direccionIP": "192.168.1.1"
      }
    - Servidor a cliente:
      { "response": "OK"}
      O bien:
      {
        "respuesta": "ERROR",
        "razon": "Nombre o dirección duplicado"
      }

2. Liberar usuario:
    - Cliente a servidor:
      {
        "tipo": "EXIT",
        "usuario": "nombre_usuario"
        "estado" : ""
      }

3. Broadcasting
    - Cliente a servidor:
      {
        "accion ": "**BROADCAST**",
        "nombre_emisor": "nombre_emisor",
        "mensaje": "mensaje"
      }
    - Servidor a todos:
      {
        "accion ": "**BROADCAST**",
        "nombre_emisor": "nombre_emisor",

"mensaje": "mensaje"
}

4. DM (Direct Message)
    - Cliente a servidor:
    {
    "accion ": "**DM**",
    "nombre_emisor": "nombre_emisor",
    "nombre_destinatario": "nombre_destinatario"
    "mensaje": "mensaje"
    }
    - Servidor a cliente:
    {
    "accion ": "**DM**",
    "nombre_emisor": "nombre_emisor",
    "nombre_destinatario": "nombre_destinatario"
    "mensaje": "mensaje"
    }
5. Solicitar lista de usuarios (conectados):
    - Cliente a servidor:
    {
    "accion ": "**LISTA**",
    "nombre_usuario": "nombre_usuario",
    }
    - Servidor a cliente:
    {
    "accion ": "**LISTA**",
    "usuarios: ["usuario1", "usuario2", "usuario3", "..."],
    }

6. Cambio de estado
    - Cliente a servidor:
    {
    “tipo”: “**ESTADO**”
    “usuario”: nombre_usuario
    “estado”: estado
    }
    - Servidor a cliente:
    Puede regresar con estado Ok o error
    {
    “respuesta”: **OK**

}

O bien:

{

“respuesta”: “**ERROR**”

“razon”: “ESTADO_YA_SELECCIONADO”

}

7. Mostrar información (ip) de un usuario (si está conectado)
    - Cliente a servidor:

      {

      “tipo”: “**MOSTRAR**”

      “usuario”: nombre_usuario

      }
    - Servidor a cliente:

      {

      “tipo”: “**MOSTRAR**”

      “usuario”: nombre_usuario

      “estado”: estado

      }

      O bien puede ser también:

      {

      “respuesta”: “**ERROR**”

      “razon”: “USUARIO_NO_ENCONTRADO”

      }

      O bien:

      {

      “respuesta”: “**ERROR**”

      “razon”: “USUARIO DESCONECTADO”

      }

¿Es necesario hacer un cliente por cada usuario?

Sí, el usuario usará un cliente para poder conectarse al servidor. La forma en la que se conecta debe ser como en el documento de instrucciones. Supongamos que el archivo se llama client.c, al ejecutarlo el script recibe los siguientes parámetros

<nombredelaaplicación> <nombredeusuario> <IPdelservidor> <puertodelservidor>

Entonces se ejecutaría de la siguiente manera utilizando argumentos como parámetros:

./client pepito 123.12.0.0 50213

El cliente intenta conectarse al server y el servidor es el encargado de revisar la conexión. La validación que se hace es que tanto el nombre como la ip no estén en el registro del servidor. Para una dirección ip no debe haber más de un nombre de usuario. Es decir, si la ip del usuario está guardada y el nombre no coincide no se permite el acceso.

Idealmente en la implementación del server se debería tener un registro de los usuarios guardados y aparte los usuarios que están conectados. Añadido a esto es importante hacer uso de threads para que el server pueda manejar conectividad de varios usuarios simultáneamente. El server debe de tener un socket al cual se le debe de asociar una dirección IP y el puerto donde estará escuchando. Librerías para tomar en cuenta

- Sys/socket.h
- Netinet/in.h
- Pthread.h
- String.h
- arpa/inet.h
- cjson/cJSON.h

Para el cliente también se tiene que hacer un socket cuando este se crea. Como idea tenemos que el cliente maneje la interfaz en la consola. Siendo una sección (la parte superior) para mostrar los mensajes recibidos y abajo el menú, por ejemplo. O bien que se guarden los mensajes y haya una opción extra para ver mensajes privados y de grupo. Pero en ambos casos probablemente el cliente necesite threads. 1 principal que escuche constantemente para esperar mensajes y otro para el input del usuario y las acciones que este puede hacer.