Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2022-1

19 de Mayo de 2022
Actividad Formativa

Actividad Formativa 3

Networking

Entrega

- **Lugar:** En su repositorio privado de GitHub, en la **carpeta** `Actividades/AF3/`
- **Hora del *push*:** 16:40

Importante: Antes de comenzar, comprueba que Git este funcionando correctamente en tu repositorio privado. Para esto, **sube los archivos base de la actividad de inmediato** (*add*, *commit*, *push*). Se espera que en esta actividad (así como en las demás actividades y tareas) utilices Git a lo largo de **todo tu desarrollo** como una herramienta, no sólo como un método de entrega. Es por esto que recomendamos enfáticamente que vayas subiendo tus cambios constantemente (***push***), ya que **problemas de último minuto** relacionados con la entrega y Git **no serán considerados**.

Introducción

Debido a la inflación, lamentablemente los servicios de streaming de música han aumentado mucho su precio. Vagando por internet, te encuentras con la solución a tus problemas: **DCCancionero**, una aplicación que te permite descargar y escuchar tu música favorita sin costo alguno, pero que ofertón. El único incoveniente es que debido a un problema de red, tu descarga de **DCCancionero** se vio corrompida y al programa le faltan líneas de código, específicamente las que manejaban la comunicación entre el cliente y el servidor. Con tus conocimientos de Networking debe ser pan comido arreglar la aplicación, por lo cual pones manos a la obra y empiezas a escribir el código que falta.

Flujo del Programa

DCCancionero es un servicio en red que se compone de dos programas: el servidor y el cliente. El servidor se encarga de almacenar y controlar el acceso a todas las canciones disponibles, junto con el manejo de todos los usuarios registrados. El cliente se encarga de interactuar directamente con el usuario, mostrando la información recibida desde el servidor y enviando solicitudes de las acciones que realiza el usuario.

Cada programa debe ejecutarse de manera **separada** y comunicarse exclusivamente mediante transmisión de datos usando sockets. Debes completar cada uno de los programas y luego probarlos en modo cliente-servidor.

Parte 0: Uso de `.gitignore`

Para esta actividad está incluido el `.gitignore` que deben utilizar para ignorar correctamente la carpeta `frontend/assets` del cliente y las carpetas `data` del cliente y del servidor.

Estructura del Programa

En el directorio de la actividad se entrega una carpeta para el servidor y otra para el cliente. Es importante recordar que los códigos correspondientes al servidor y al cliente son **excluyentes**.

```
servidor
├── main.py
├── logica.py
├── servidor.py  Debes modificarlo
├── utils.py
└── data
    ├── caratulas
    │   └── .jpg
    └── musica
        └── .wav

cliente
├── main.py
├── frontend
│   ├── ventana_carga.py
│   ├── ventana_login.py
│   ├── ventana_principal.py
│   └── assets
│       └── archivos .ui y .png
├── backend
│   ├── cliente.py  Debes modificarlo
│   └── interfaz.py
├── data
│   ├── musica/
│   └── caratulas/
├── parametros.json
└── utils.py
```

Servidor

El servidor está implementado en la carpeta `servidor/`, la cual contiene los siguientes archivos:

- `main.py`: Archivo principal del servidor. Esté instancia la clase Servidor e inicia su funcionamiento para aceptar clientes. No debes modificarlo **No debes modificarlo**
- `logica.py`: Contiene las funciones necesarias para la lógica del servidor, tanto para el manejo de archivos como para el procesado de señales intercambiadas con los clientes **No debes modificarlo**
- `servidor.py`: Contiene a la clase Servidor, con los atributos y métodos necesarios para establecer una correcta comunicación con el cliente. **Debes modificarlo**
- `utils.py` : Contiene funciones para la lectura y escritura de archivos. **No debes modificarlo**
- `data/`: Es la carpeta que almacena en el servidor todos los archivos que se envían al cliente cuando es solicitado. **No debes modificarlo**

Parte I: Implementación del servidor

En esta parte deberás completar la clase `Servidor` del archivo `servidor.py`, la cual posee los siguientes métodos:

- `def __init__(self, host: str, port: int) -> None`: Inicializa el servidor para recibir conexiones. Posee los siguientes atributos: **No debes modificarlo**
  - `self.host` : Es un `str` que representa la dirección del servidor.
  - `self.port`: Es un `int` que representa el puerto del servidor
  - `self.socket_servidor`: Es una instancia de `socket` que acepta las conexiones de los clientes. Inicialmente se encuentra inicializado en `None`, por lo que deberás asignarle su nuevo valor en el método `iniciar_servidor`.
  - `self.logica`: Es una instancia de la clase `Logica`, la cual actúa como el *back-end* del servidor
  - `self.clientes_conectados`: Es un `dict` que contiene la información sobre los clientes conectados, donde cada llave corresponde al id de un cliente y cada valor corresponde a su respectivo `socket`.
- `def iniciar_servidor(self) -> None`: Este método se encarga de inicializar al servidor para comenzar a escuchar conexiones. Para esto, primero debes crear el socket del servidor con dirección IPv4 y protocolo de transporte TCP. Debes guardar el socket en el atributo `self.socket_servidor`. Luego, debes enlazar el host en el que está corriendo el servidor al puerto donde se quiere escuchar las conexiones, dados por los atributos `self.host` y `self.port` respectivamente. Después, debes hacer que el servidor empiece a escuchar conexiones y cambiar su estado a conectado. Finalmente, mediante el método `log` debes imprimir un mensaje que indique el host y el puerto donde el servidor está escuchando y llamar al método `comenzar_aceptar`. Este método no retorna nada. **Debes modificarlo**
- `def comenzar_a_aceptar(self) -> None`: En este método debes instanciar e iniciar un daemon `thread` que ejecutará el método `aceptar_clientes` para ir aceptando y estableciendo conexiones con todos los clientes que lo soliciten. Este método no retorna nada. **Debes modificarlo**
- `def aceptar_clientes(self) -> None`: Este método deberá estar constantemente esperando y aceptando conexiones de clientes y encargarse de que estos a su vez sean escuchados. Para esto, en primer lugar debes aceptar la conexión con el cliente entrante y recuperar su instancia de socket

correspondiente. Si no se logra establecer la conexión, debes manejar el error y terminar el funcionamiento del método. Finalmente, debes instanciar e iniciar un nuevo Thread sólo para el cliente aceptado, el cual estará encargado de escucharlo mediante el método `escuchar_cliente`. Recuerda que este método recibe el id del cliente a escuchar, por lo que debes encargarte de modificar el atributo `self.id_cliente` por cada cliente aceptado. **Debes modificarlo**

- `def escuchar_cliente(self, id_cliente: int, socket_cliente: socket) -> None`: Este método se encarga de escuchar constantemente al cliente asignado a través de su socket. Para esto, primero debes esperar constantemente un mensaje del cliente utilizando el método `recibir_mensaje`. Si este mensaje está vacío, debes levantar un error de conexión que deberá hacer un llamado a `eliminar_cliente`. Luego deberás procesar una respuesta para este mensaje utilizando el método `procesar_mensaje` proveniente de la clase lógica ya instanciada. Enseguida, debes verificar si existe respuesta, es decir, que `procesar_mensaje` no retorne un dict vacío para finalmente enviar la respuesta al cliente utilizando el método `enviar_mensaje`. Si en algún momento surge un error de conexión, se deberá hacer un llamado a `eliminar_cliente`. **Debes modificarlo**
- `def recibir_mensaje(self, socket_cliente: socket) -> dict:`
  Aquí debes implementar un método que administre la recepción del mensaje utilizando el siguiente protocolo para decodificar y retornar el mensaje: **Debes modificarlo**
  1. Se recibe el largo del mensaje, especificado en los primeros 4 bytes recibidos ocupando byteorder big endian (`"little"`).
  2. El mensaje se debe leer por partes, para ello, debes leer `chunks` de `64 bytes`, hasta que se haya leído el mensaje completo
  3. Se decodifica el mensaje recibido completo utilizando el método `decodificar_mensaje`. Este resultado es el que se debe retornar.
- `def enviar_mensaje(self, mensaje: dict, socket_cliente: socket) -> None:`
  Este método se encarga de enviar mensajes al cliente, utilizando el mismo protocolo de comunicación mencionado anteriormente. Primero, el mensaje debe ser codificado usando el método `codificar_mensaje`. Luego, debes obtener el largo del mensaje en 4 bytes y ocupar el byteorder little (`"little"`). Por último, debes enviar el largo del mensaje junto al mensaje codificado (en ese orden) al cliente. Este método no retorna nada, sino que usa el metodo `send` del socket para comunicarse. **Debes modificarlo**
- `def enviar_archivo(self, socket_cliente: socket, ruta: string`: Este método se encarga de leer el archivo especificado por ruta y luego enviarlo al cliente en forma de `chunks` **No debes modificarlo**
- `def eliminar_cliente(self, id_cliente: int, socket_cliente: socket)`: Se encarga de eliminar un cliente del diccionario de clientes conectados **No debes modificarlo**
- `def codificar_mensaje(self, mensaje: dict) -> bytes`: En este método debes serializar y codificar un mensaje usando JSON, para finalmente retornarlo. **Debes modificarlo**
- `def decodificar_mensaje(self, mensaje_bytes: bytes) -> dict`: En este método debes deserializar y decodificar un mensaje usando JSON, para finalmente retornarlo. **Debes modificarlo**

Cliente

El cliente esta implementado en la carpeta `cliente/`, la cual contiene los siguientes archivos y directorios:

- `main.py`: Archivo principal del cliente. Este instancia a la clase `Cliente` que se conecta con el servidor para interactuar con el programa . No debes modificarlo **No debes modificarlo**

- `frontend/`: Es la carpeta del cliente que contiene los módulos `ventana_carga`, `ventana_login` y `ventana_principal`, los cuales son necesarios para mostrar la interfaz gráfica del cliente. No debes modificarlo
- `backend/cliente`: Contiene la clase `Cliente` con los atributos y métodos necesarios para establecer una conexión y comunicarse con el servidor. Debes modificarlo
- `backend/interfaz`: Contiene la clase `Interfaz` con los métodos necesarios para mostrar la interfaz gráfica y hacer las conexiones entre el backend y frontend del cliente. No debes modificarlo
- `data/`: Es la carpeta en la cual se van a guardar las canciones descargadas del servidor. No debes modificarlo
- `utils.py` : Contiene funciones para la lectura y escritura de archivos. No debes modificarlo

Parte II: Implementación del cliente

En esta parte deberás completar la clase `Cliente` del archivo `cliente.py`, la cual posee los siguientes métodos:

- `def __init__(self, host: str, port: int) -> None`: Inicializa el cliente, creando un `socket` y conectándolo al servidor. Posee los siguientes atributos: No debes modificarlo
  - `self.host`: Es un `str` que corresponde a la dirección del servidor
  - `self.port`: Es un `int` que corresponde al puerto en el que está escuchando el servidor.
  - `self.socket_cliente`: Es una instancia de `socket` con la que el cliente se conecta al servidor.
  - `self.conectado`: Es un `bool` que es `True` si el cliente esta conectado al servidor y `False` en caso contrario.
  - `self.interfaz`: Es una instancia de la clase `Interfaz`, la cual se encarga de manejar la interfaz gráfica del cliente.
- `def iniciar_clientes(self) -> None`: Este método se encarga de inicializar el cliente. Para esto, primero debes conectar el socket del cliente al servidor con protocolo TCP. Recuerda que en este paso puede ocurrir un error de conexión, por lo que debes manejar esta excepción adecuadamente para asegurar el correcto funcionamiento del programa. En caso que ocurra este error, debes imprimir un mensaje mediante el método `log` que indique el error. Luego, debes llamar al método `comenzar_a_escuchar` y al método `mostrar_ventana_carga` de la clase `Interfaz`. Debes modificarlo
- `def comenzar_a_escuchar(self) -> None`: En este método debes crear e iniciar un daemon thread que se encargue de ejecutar el método `escuchar_servidor`. Debes modificarlo
- `def escuchar_servidor(self) -> None`: Este método se encarga de escuchar **constantemente** los mensajes enviados por el servidor **mientras el cliente se encuentre conectado**, para lo cual debes hacer uso del método `recibir_mensaje`. Al implementar esto, ten en cuenta que se puede producir un error de conexión, por lo que debes manejar esta excepción adecuadamente. Si se recibe un mensaje, el cliente debe procesarlo mediante el método `manejar_mensaje` de la clase `Interfaz`. Debes modificarlo
- `def recibir(self) -> dict`: Este método se encarga de recibir los mensajes enviados por el servidor. Para esto, primero debes recibir el largo del mensaje en 4 *bytes* que fue previamente serializado en *little endian*. Luego, debes recibir la información en chunks de máximo 64 *bytes* cada uno. Por último, debes decodificar el mensaje utilizando el método `decodificar_mensaje` y retornarlo. Debes modificarlo

- `def enviar(self, mensaje: dict) -> None`: Este método se encarga de enviar un mensaje al servidor. Para esto, debes codificar el mensaje utilizando el método `codificar_mensaje`. Luego, debes obtener el largo del mensaje en 4 *bytes* y serializarlo en *little endian*. Por último, debes hacer envío del largo del mensaje junto con el mensaje codificado, todo en un sólo envío. **Debes modificarlo**
- `def codificar_mensaje(self, mensaje: dict) -> bytes`: En este método debes serializar y codificar un mensaje usando JSON, para finalmente retornarlo. **Debes modificarlo**
- `def decodificar_mensaje(self, mensaje_bytes: bytes) -> dict`: En este método debes deserializar y decodificar un mensaje usando JSON, para finalmente retornarlo. **Debes modificarlo**

Notas

- Se recomienda fuertemente para esta actividad que tanto el programa de cliente como el de servidor se ejecuten **directamente en la consola de tu sistema**, con el motivo de evitar problemas que puedan generar los editores de texto.
- **Recuerda correr primero el servidor y luego el cliente**, además de reiniciar el servidor cada vez que se hagan cambios en el código de este.

Objetivos

- Implementar servidor el cual sea capaz de recibir, manejar y enviar mensajes a múltiples clientes conectados de forma simultánea.
- Implementar cliente capaz de conectarse y comunicarse con un servidor.

Anexo

0.1. Comandos del servidor

Comandos que recibe el servidor:

- **validar_login** : se le pide al servidor validar si el nombre de usuario ingresado no esta aún registrado y que la contraseña sea igual a la de `parametros.json`.
- **descargar_musica** : se pide al servidor que envie la data correspondiente a la canción seleccionada.
- **desconectar** : se elimina un nombre de usuario del servidor cuando este se desconecta.

Requerimientos

- (0.5 pts) Parte I : Implementación del Servidor
- (0.5 pts) Parte II : Implementación del Cliente