Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2021-1

22 de junio de 2021
Actividad Formativa

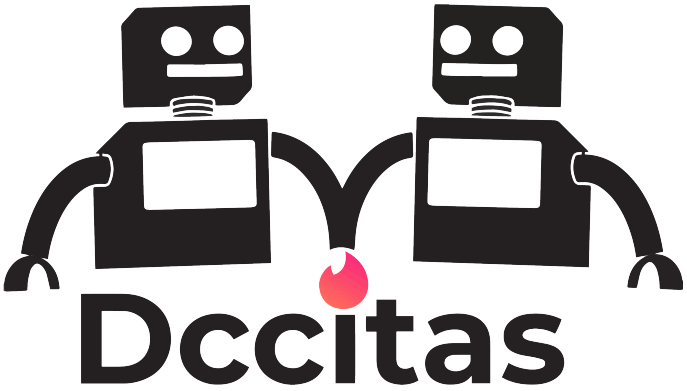
Actividad Formativa 7

Networking

Entrega

- **Lugar:** En su repositorio privado de GitHub, en la **carpeta** `Actividades/AF7/`
- **Hora del *push*:** 20:00, miércoles 23 de junio

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

Al estar tanto tiempo en la soledad del encierro debido a la cuarentena, esperas con ansias volver a salir y conocer gente. Por mientras, decides poner a buen uso todos tus conocimientos de *networking* para desarrollar **DCCitas**, una *app* que permite a sus usuarios tener un perfil, hacer *match* y chatear con otras personas. El mecanismo para hacer *match* con otros usuarios de la plataforma puede parecerte bastante familiar. Se muestran perfiles aleatorios, y puedes elegir a quién darle *like*, y a quién pasar. Si la misma persona te da *like* de vuelta, ¡es un *match*!. Para lograrlo, deberás establecer la conexión e implementar un intercambio de mensajes entre un servidor y múltiples clientes mediante el uso de *sockets*.

Flujo del Programa

La implementación de **DCCitas** se compone de dos partes: el **servidor** y el **cliente**. El servidor se encarga de mantener y acceder a los datos, y manejar toda la lógica de los perfiles, los *likes*, y los *matches*. El cliente se encarga de interactuar directamente con el usuario (mediante una interfaz gráfica), mostrar la información recibida desde el servidor, y enviar de vuelta cambios y acciones que el usuario quiera realizar. Ambas partes se ejecutan como procesos totalmente independientes (de hecho, podrían ejecutarse en computadores distintos), y su única comunicación ocurre a través de los *sockets*.

Estructura del Programa

En el directorio de la actividad se te entrega una carpeta para el cliente y otra para el servidor, las cuales contienen los archivos necesarios para su ejecución. Cabe destacar que el código del servidor y el código del cliente son completamente independientes entre sí, y la única interacción entre ellos se produce por medio de *sockets*. El contenido del directorio con el cual trabajarás en esta actividad se detalla a continuación:

```
AF7/
├── servidor/
│   ├── db/
│   │   ├── images/
│   │   ├── likes.csv
│   │   ├── mensajes.csv
│   │   └── usuarios.json
│   ├── logica.py
│   ├── main.py
│   ├── servidor.py ## <--- Parte I
│   └── usuario.py
└── cliente/
    ├── assets/
    ├── ventanas/
    ├── cliente.py ## <--- Parte II
    ├── interfaz.py
    └── main.py
```

Servidor

El servidor está contenido en la carpeta `servidor/`, que contiene los siguientes archivos:

- `db/`: Carpeta que contiene toda la base de datos para que el programa funcione, en donde se encuentran los siguientes archivos:
  - `images/`: Carpeta que contiene las imágenes de los perfiles de usuarios.
  - `likes.csv`: Archivo en donde la primera columna corresponde al usuario que da el *like* y la segunda corresponde al usuario que recibe el *like*.
  - `mensajes.csv`: Archivo en donde se almacenarán los mensajes enviados.
  - `usuarios.json`: Archivo que contiene a todos los usuarios de la aplicación, con su nombre, edad y biografía respectiva.
- `logica.py`: Contiene a la clase `Logica`, que implementa la lógica y el flujo del programa.

- `main.py`: Corresponde al archivo principal del servidor. Este instancia la clase `Servidor` e inicia su funcionamiento para aceptar conexiones de clientes.
- `servidor.py`: Contiene a la clase `Servidor`, con los métodos y atributos necesarios para establecer una correcta comunicación con el cliente.
- `usuario.py`: Contiene la clase `Usuario`, que le permite al servidor guardar la información de los clientes que se conecten, para poder guardar sus atributos y comunicarse con ellos.

Parte I

El archivo donde deberás trabajar corresponde a `servidor.py`, en la clase `Servidor`.

Sin embargo, en el archivo `logica.py` se encuentra la clase `Logica` que ya viene implementada, pero tiene un método que debes utilizar en esta parte:

- `def manejar_mensaje(mensaje: dict, id_cliente: int) - > (respuesta: dict, ids: list)`: recibe el mensaje decodificado y el id del cliente que lo envió. Retorna una respuesta y una lista con ids: donde el primer elemento es el id del cliente que envió el mensaje, y el segundo es el id del cliente objetivo de la respuesta. Este método ya viene implementado, por lo que **No debes modificarlo**, pero debes usarlo para manejar el mensaje recibido en el método `escuchar_cliente` de la clase `Servidor`.

A continuación se describen los métodos de la clase `Servidor` que **ya están implementados y NO deben ser modificados**:

- `def __init__(self, host, port, log_activado=True)`: Inicializa el servidor. 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.log_activado`: Es un `bool` que indica si la funcionalidad de imprimir mensajes en la consola se encuentra activa o no.
  - `self.clientes_conectados`: Es un `dict` que contiene información sobre los clientes conectados, en donde cada llave corresponde al id de un cliente y cada valor corresponde a su respectivo socket.
  - `self.logica`: Es una instancia de la clase `Logica`, la cual actúa como el *back-end* del servidor
  - `self.socket_servidor`: Es una instancia de `socket`, que acepta las conexiones desde los clientes, y funciona como el punto de conexión inicial con los clientes, y su propósito es aceptar conexiones. Deberás asignarle su nuevo valor en el método `iniciar_servidor`.
- `def log(self, mensaje_consola: str)`: Imprime un mensaje en consola, si la funcionalidad está activada. **No debes modificarlo**
- `def eliminar_cliente(self, id_cliente: int)`: Elimina un cliente que se encuentra conectado al servidor. **No debes modificarlo**
- `def cerrar_servidor(self)`: Cierra el servidor al cerrar su *socket* y desconecta a los clientes. **No debes modificarlo**
- `def codificar_mensaje(self, mensaje : dict) -> bytes`: Serializa y codifica un mensaje usando JSON. **No debes modificarlo**

- `def decodificar_mensaje(self, bytes_mensaje: bytes) -> dict`: Decodifica y deserializa un mensaje usando JSON. **No debes modificarlo**

Por otra parte, los métodos que se describen a continuación **DEBEN ser implementados:**

- `def iniciar_servidor(self):` Este método se encarga de inicializar el servidor. Para esto, primero debes crear el socket del servidor con dirección IPv4 y protocolo de transporte TCP. El socket **debe** ser guardado 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. Finalmente, debes hacer que el servidor empiece a escuchar conexiones. **Este método no retorna nada.** **Debes modificarlo**
- `def aceptar_clientes(self):` En primer lugar, debes hacer que este método acepte **constantemente** conexiones de clientes a través del socket del servidor. Luego, por cada cliente que sea aceptado, debes imprimir un mensaje mediante el método `log` que indique esto, de la forma: `f"Un nuevo cliente con dirección {direccion_ip} ha sido aceptado"`. Además, una vez que un cliente haya sido aceptado, este debe ser agregado al diccionario `self.clientes_conectados`. En este paso, para definir el id del cliente debes usar el valor del atributo de clase `_id_cliente`, el cual debes ir actualizando, de tal forma que estos no se repitan. Para agregar un cliente al diccionario debes tener en cuenta que puede ocurrir que múltiples clientes se intenten conectar al mismo tiempo, por lo que debes implementar un mecanismo que asegure el correcto manejo del diccionario. Por último, cada vez que un cliente se conecte debes hacer que este comience a recibir información de inmediato, utilizando el método `escuchar_cliente`. Recuerda que el servidor debe poder escuchar y conectar a múltiples clientes simultáneamente. **Este método no retorna nada.** **Debes modificarlo**
- `def escuchar_cliente(self, id_cliente: int):` Este método se encarga de escuchar **constantemente** los mensajes enviados por un cliente, para lo cual debes hacer uso del método `recibir`. Al implementar esto, ten en cuenta que se puede producir un error de conexión (`ConnectionResetError`) que provoca que el método `recibir` retorne un mensaje vacío[1], por lo que debes manejar esta excepción adecuadamente. En caso que se levante una excepción, debes imprimir un mensaje mediante el método `log` que indique que se produjo un error de conexión con el cliente, de la forma: `f"ERROR: conexión con cliente {id_cliente} fue reseteada"`, y después eliminar al cliente mediante el método `eliminar_cliente`. En caso que se reciba un mensaje del cliente, el servidor debe generar una respuesta apropiada mediante el método `manejar_mensaje` de la clase `Logica`. Finalmente, a partir de lo obtenido en el paso anterior, deberás enviar la respuesta a los destinatarios correspondientes mediante el método `enviar`. **Este método no retorna nada.** **Debes modificarlo**
- `def enviar(self, mensaje: dict, socket_cliente: socket):` Este método se encarga de enviar un mensaje a un cliente. Para esto, debes codificar el mensaje utilizando el método `codificar_mensaje`. Luego, debes obtener el largo de este mensaje en 5 *bytes* y serializar en *little endian*. Por último, debes hacer envío del largo de mensaje junto con el mensaje codificado al cliente respectivo, todo en un solo envío. Este método no retorna nada. **Debes modificarlo**
- `def recibir(self, socket_cliente: socket) -> dict:` Este método se encarga de recibir los mensajes enviados por un cliente. Para esto, debes recibir el largo del mensaje en 5 *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 el diccionario respectivo. **Debes modificarlo**

[1] En plataformas UNIX (Linux y MacOS), al cerrarse abruptamente la conexión con en cliente, el *socket* recibe contínuamente un string vacío, por lo que hacer este raise permitiría manejar este caso.

Cliente

El cliente está implementado en la carpeta `cliente/`, y tiene los siguientes archivos:

- `assets/`: Contiene los elementos visuales (imágenes) de la interfaz gráfica del cliente.
- `ventanas/`: Contiene los archivos correspondientes a la implementación de la interfaz gráfica.
- `cliente.py`: Contiene a la clase `Cliente`, con los métodos y atributos necesarios para establecer una correcta comunicación con el servidor.
- `interfaz.py`: Contiene la clase `Controlador`, que se encarga de coordinar la interfaz gráfica con respecto a las instrucciones del cliente y del servidor.
- `main.py`: Corresponde al archivo principal del cliente, que instancia la clase Cliente que a su vez se conecta al servidor para poder interactuar con el programa.

Parte II

El archivo donde deberás trabajar corresponde a `cliente.py`, en la clase `Cliente`.

A continuación se describen los métodos de la clase `Cliente` que **ya están implementados y NO deben ser modificados**:

- `def __init__(self, host, port, log_activado=True)`: Inicializa el cliente, creando un socket y conectándolo al servidor. Posee los siguientes atributos: No debes modificarlo
  - `self.host`: Es un `str` que tiene la dirección del servidor.
  - `self.port`: Es un `int` que tiene el número de puerto al que se va a conectar.
  - `self.log_activado`: Es un `bool` que indica si la funcionalidad de imprimir mensajes en la consola se encuentra activa o no.
  - `self.controlador`: Es una instancia de la clase `Controlador`, que se encarga de manejar la interfaz gráfica del cliente.
  - `self.socket_cliente`: Es el `socket` del cliente, con el que se conecta al servidor.
- `def log(self, mensaje_consola)`: Imprime un mensaje en consola, si la funcionalidad está activada. No debes modificarlo
- `def codificar_mensaje(self, mensaje : dict) -> bytes`: Codifica y serializa un mensaje usando JSON. No debes modificarlo
- `def decodificar_mensaje(self, bytes_mensaje: bytes) -> dict`: Decodifica y deserializa un mensaje usando JSON. No debes modificarlo

Por otra parte, los métodos que se describen a continuación **DEBEN ser implementados:**

- `def iniciar_cliente(self)`: 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 que no se pudo establecer la conexión, de la forma `f"No se pudo conectar a {self.host}:{self.port}"`, y después cerrar el socket de cliente. Luego, si esta conexión se realiza de forma exitosa debes definir el atributo `self.conectado` como `True`. Finalmente, debes hacer que el cliente empiece a escuchar **constantemente** los mensajes que

envíe el servidor y mostrar la interfaz utilizando el método `mostrar_login` de `self.controlador` (no recibe argumentos). **Este método no retorna nada.** Debes modificarlo

- `def escuchar_servidor(self)`: 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`. Al implementar esto, ten en cuenta que se puede producir un error de conexión (`ConnectionResetError`), por lo que debes manejar esta excepción adecuadamente. Tal como ocurre con el servidor, deberás levantar la excepción `ConnectionResetError` en caso de recibir un mensaje vacío. En caso que se levante este error, debes imprimir un mensaje mediante el método `log` que indique el error que se produjo, de la forma: `"Error de conexión con el servidor"`. En caso que se reciba un mensaje del servidor, el cliente debe procesarlo mediante el método `manejar_mensaje` de la clase `Controlador`. Finalmente, una vez que se termine la conexión con el servidor, debes encargarte de cerrar el socket del cliente. **Este método no retorna nada.** Debes modificarlo
- `def enviar(self, mensaje: dict)`: Este método se encarga de enviar un mensaje al servidor. Para esto, debes codificar el mensaje utilizando el método `self.codificar_mensaje`. Luego, debes obtener el largo de este mensaje en 5 *bytes* y serializar en *little endian*. Por último, debes hacer envío del largo del mensaje junto con el mensaje codificado, todo en un sólo envío. Al implementar este último paso, ten en cuenta que se puede producir un error de conexión, específicamente un `ConnectionError`, por lo que este debe ser manejado adecuadamente y cerrar el socket del cliente. **Este método no retorna nada.** Debes modificarlo
- `def recibir(self) -> dict`: Este método se encarga de recibir los mensajes enviados por el servidor. Para esto, debes recibir el largo del mensaje en 5 *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

Notas

- El código que escribas en la parte I puede servirte para la parte II. Si bien no será directamente un *copy-paste*, se te hará mucho más fácil hacer el ítem equivalente en el cliente si te basas en el código que escribiste en servidor.
- Para esta actividad te recomendamos ejecutar los programas de cliente y servidor **directamente en la consola de tu sistema**, esto para evitar problemas que puedan generar los editores.
- Recuerda reiniciar el servidor cada vez que hagas algún cambio para hacerlos efectivos.
- Son libres de crear nuevos atributos y métodos que crean necesarios para el desarrollo de sus programas.
- Para esta actividad la corrección será automática. Esto quiere decir que el punyaje dependerá de si las funcionalidades están implementadas correctamente o no.

Objetivos

- Implementar servidor capaz de recibir, manejar y enviar mensajes a múltiples clientes
- Implementar cliente capaz de comunicarse y conectarse a un servidor