Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2021-2

11 de Noviembre de 2021
Actividad Formativa

Actividad Formativa 4

Networking + I/O

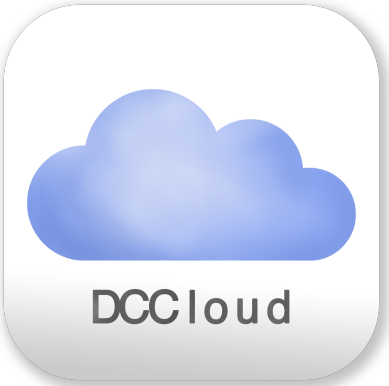
Entrega

- **Lugar:** En su repositorio privado de GitHub, en la **carpeta** `Actividades/AF4/`
- **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

¿Alguna vez te has preguntado cómo es que entre profes se pasan memes, soluciones de Interrogaciones o los PPT de un semestre a otro? **DCCloud** es la aplicación oficial para que profesoras y profesores de la Universidad puedan intercambiar archivos **ultra secretos**. Sin embargo, un grupo de *hackers* modificaron el código del programa para poder acceder a toda la información **ultra secreta**... ahora la privacidad de los profes corre grave peligro. Afortunadamente el DCC cuenta contigo y tus conocimientos de *networking* e *Input/Output* (I/O) para reescribir **DCCloud** y poder retomar el envío seguro de los archivos **ultra secretos**.



Flujo del Programa

DCCloud 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 todos los archivos de los usuarios, además de manejar toda la lógica de la nube. El **cliente** se encarga de interactuar directamente con el usuario, mostrando la información recibida desde el servidor y enviando solicitudes de las acciones disponibles para el usuario: explorar, almacenar, subir y descargar archivos.

Cada programa debe ejecutarse de manera separada y comunicarse exclusivamente mediante transmisión de datos por la red usando *sockets*. Debes completar cada uno de los programas y luego probarlos en modo cliente-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 y se ejecutan de manera separada entre sí, por lo que la única interacción entre ambos es mediante *sockets*.

```
AF4/
├── servidor/
│   ├── data/
│   │   ├── fotos/
│   │   ├── gifs/
│   │   └── sonidos/
│   ├── main.py
│   ├── manejo_archivos.py
│   ├── parametros.py
│   └── servidor.py ## <--- Parte I
└── cliente/
    ├── ascii/
    ├── data/
    │   ├── fotos/
    │   ├── gifs/
    │   └── sonidos/
    ├── archivos.py
    ├── cliente.py ## <--- Parte II
    ├── main.py
    ├── menus.py
    └── parametros.py
```

Servidor

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

- `data/`: Es la carpeta que almacena en el servidor todos los archivos subidos a la nube. Dentro de esta se encuentran los subdirectorios `fotos/`, `gifs/` y `sonidos/`. **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**
- `main.py`: Archivo principal del servidor. Este instancia la clase `Servidor` e inicia su funcionamiento para aceptar clientes. **No debes modificarlo**

- `manejo_archivos.py`: Contiene las funciones necesarias para la lógica de almacenamiento en la nube. No debes modificarlo
- `parametros.py`: Contiene todos los parámetros necesarios para el funcionamiento del servidor. No debes modificarlo

Parte I: Implementación del servidor

Debes desarrollar esta parte en la clase `Servidor` dentro de `servidor/servidor.py`. Los siguientes métodos **ya se encuentran implementados y NO debes modificarlos**.

- `def __init__(self, host: str, port: int):` No debes modificarlo
  Inicializa y prepara al servidor para recibir conexiones. Define los atributos:
  - `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 desde los clientes.
  - `self.clientes_conectados`: Es un `dict` que contiene información sobre los clientes conectados, donde cada llave corresponde al id único de un cliente (asignado mediante el atributo `_id_cliente`, un atributo de clase de tipo `int` que va aumentando al establecer cada conexión) y cada valor corresponde a su respectivo `socket`.
- `def manejar_comando(self, recibido: dict, socket_cliente: socket) -> dict:` No debes modificarlo
  Recibe los mensajes decodificados (en forma de diccionario) enviados por algún cliente, y el *socket* correspondiente. Identifica la solicitud del cliente (explorar, almacenar, subir o descargar) y ejecuta las acciones necesarias para responder a la solicitud. El metodo retorna un `dict`.
- `def enviar_archivo(self, socket_cliente: socket, ruta: str):` No debes modificarlo
  Recibe una ruta de un archivo a enviar. Separa el archivo en *chunks* de 8000 *bytes* cada uno y los envía usando el método `enviar`.
- `def codificar_mensaje(self, mensaje: dict) -> bytes:` No debes modificarlo
  Serializa y codifica un mensaje utilizando JSON.
- `def decodificar_mensaje(self, msg_bytes: bytes) -> dict:` No debes modificarlo
  Decodifica y deserializa un mensaje recibido, utilizando JSON.

Por otro lado, los métodos que se describen a continuación **DEBEN ser implementados:**

- `def unir_y_escuchar(self):` Debes modificarlo
  Aquí debes asociar al `socket` al puerto correspondientes estableciendo una conexión de tipo **TCP**, y hacer que se comiencen a escuchar conexiones entrantes. Luego, debes imprimir un mensaje que indique el `host` y el puerto donde se está escuchando. Finalmente debes llamar al método `aceptar_conexiones` para comenzar a aceptar las conexiones. **Este método no retorna nada**.
- `def aceptar_conexiones(self):` Debes modificarlo
  Aquí debes instanciar e iniciar un `Thread` que ejecutará el método `thread_aceptar_conexiones` para ir aceptando y estableciendo conexiones con todos los clientes que lo soliciten. **Este método no retorna nada**.
- `def thread_aceptar_conexiones(self):` Debes modificarlo
  Este método deberá estar constantemente esperando y aceptando conexiones de clientes y encargarse de que estos a su vez sean escuchados constantemente. 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. Luego, debes guardar esta instancia `socket` en el diccionario `clientes_conectados` siguiendo el formato indicado anteriormente (recuerda aumentar en 1 el valor de `_id_cliente` después de utilizar ese valor). Finalmente, debes instanciar e iniciar un nuevo `Thread` solo para el cliente aceptado, el cual estará encargado de escucharlo constantemente mediante el método `thread_escuchar_cliente`. **Este método no retorna nada**.

- `def thread_escuchar_cliente(self, socket_cliente: socket, id_cliente: int):` **Debes modificarlo**
  Este método se encarga de escuchar constantemente al cliente asignado (`socket_cliente`). Para esto, primero debes esperar constantemente el mensaje decodificado del cliente utilizando el método `recibir_mensaje`. Si este mensaje está vacío, debes levantar un error de conexión.Luego deberás procesar una respuesta para este mensaje utilizando el método `manejar_comando`. Enseguida, debes verificar si existe respuesta, es decir, que `manejar_comando` no retorne un `dict` vacío para finalmente enviar la respuesta al cliente utilizando el método `enviar`. Si en algún momento surge un error de conexión, debes cerrar el `socket` del cliente y notificar esto mediante un mensaje en consola. **Este método no retorna nada**.

- `def recibir_mensaje(self, socket_cliente: socket) -> dict:` **Debes modificarlo**
  Aquí debes implementar un método que administre la recepción del mensaje utilizando el siguiente protocolo para decodificar y retornar el mensaje:

  1. Se recibe el largo del mensaje, especificado en los primeros 4 bytes recibidos ocupando `byteorder` *big endian* (`"big"`)[1].
  2. Se recibe el mensaje en *chunks* de máximo 4096 *bytes* cada uno.
  3. Se decodifica el mensaje recibido completo utilizando el método `decodificar_mensaje`. Este resultado es el que se debe retornar.

- `def enviar(self, mensaje: dict, sock_cliente: socket):` **Debes modificarlo**
  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` *big endian* (`"big"`). Por último, debes enviar el largo del mensaje junto al mensaje codificado (en ese orden) al cliente. **Este método no retorna nada**.

Cliente

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

- `data/`: Es la carpeta del cliente que guarda los archivos locales. En ella se encuentran las siguientes subcarpetas `fotos/`, `gifs/` y `sonidos/`. **No debes modificarlo**
- `cliente.py`: Contiene la clase `Cliente` con los atributos y métodos necesarios para establecer conexión y comunicarse con el servidor. **Debes modificarlo**
- `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**
- `menus.py`: Contiene las funciones que permiten la interacción del usuario con el cliente en consola. **No debes modificarlo**

[1] El argumento `byteorder` determina el orden de los bytes que representan a un `int`, y puede ser *big endian* (`"big"`) o *little endian* (`"little"`). Más información aquí.

- `archivos.py`: Contiene las funciones necesarias para acceder y guardar archivos localmente. No debes modificarlo
- `ascii/`: Dentro de esta carpeta se encuentra el archivo `title.txt` que contiene el *header* de la app **DCCloud** en arte ASCII. No debes modificarlo
- `parametros.py`: Contiene todos los parámetros necesarios para el funcionamiento del cliente. No debes modificarlo

Parte II: Implementación del cliente

El archivo donde debes desarrollar esta parte es `cliente/cliente.py`, específicamente en la clase `Cliente`. Los siguientes métodos de la clase `Cliente` **ya se encuentran implementados y NO debes modificarlos**.

- `def __init__(self, host: str, port: int):` No debes modificarlo
  Inicializa el cliente, creando un `socket` y conectándolo al servidor. Algunos atributos importantes son:
  - `self.host`: Es un `str` que representa la dirección del servidor.
  - `self.port`: Es un `int` que representa el número del puerto al que se va a conectar.
  - `self.socket_cliente`: Es la instancia `socket` del cliente con el cual se conecta al servidor.
- `def recibir_input(self):` No debes modificarlo
  Captura los *inputs* del usuario para generar la solicitud del cliente que será enviada al servidor. Esto lo hace instanciando e inicializando un `Thread` que ejecuta el método `thread_recibir_input`.
- `def manejar_comando(self, recibido: dict):` No debes modificarlo
  Maneja las respuestas que envía el servidor hacia el cliente.
- `def codificar_mensaje(mensaje: dict) -> bytes:` No debes modificarlo
  Serializa y codifica un mensaje utilizando JSON.
- `def decodificar_mensaje(msg_bytes: bytes) -> dict:` No debes modificarlo
  Decodifica y deserializa un mensaje utilizando JSON.
- `def thread_recibir_input (self):` No debes modificarlo
  Se encarga de recibir y administrar los inputs que entrega el cliente.
- `def salir(self):` No debes modificarlo
  Cierra la conexión y termina el código.
- `def print_espacio (msg: str):` No debes modificarlo
  Imprime cuanto almacenamiento esta ocupado.
- `def escribir_descarga (self):` No debes modificarlo
  Escribe la informacion almacenada en `self.descarga_actual` en el archivo correspondiente.
- `def explorar_archivos(self, recibido:str, descargar: bool):` No debes modificarlo
  Si descargar es `False` simplemente imprime los nombres de los archivos del servidor. Si descargar es `True` la funcion llama a `menus.menu_descarga` para administar la descarga del archivo deseado del servidor.
- `def salir(self):` No debes modificarlo
  Cierra el `socket` del cliente y termina el programa.

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

- `def conectar_al_servidor(self):` **Debes modificarlo**
  Este método crea la conexión del cliente con el servidor a través del `host` y `port` específicos. Además debe imprimir en consola:

```
"El cliente se ha conectado exitosamente al servidor"
```

  Este método no retorna nada.

- `def escuchar(self):` **Debes modificarlo**
  Este método instancia e inicia el `Thread` que escuchará los mensajes del servidor utilizando método `thread_escuchar`. **Este método no retorna nada**.

- `def thread_escuchar(self):` **Debes modificarlo**
  Este método se encarga de escuchar **constantemente** los mensajes enviados por el servidor **mientras el cliente se encuentre conectado**. Para esto, debes recibir el largo del mensaje en los primeros 4 *bytes* (representado en *big endian*). Luego debes recibir la información en chunks de 4096 *bytes* hasta extraer el contenido completo del mensaje (según el largo obtenido anteriormente). Una vez obtenido todo el mensaje debes decodificarlo usando el método `decodificar_mensaje` y entregar esta respuesta al método `manejar_comando` para procesarla. **Este método no retorna nada**.

- `def enviar(self, msg: dict):` **Debes modificarlo**
  Este método se encarga de enviar mensajes al servidor, utilizando el mismo protocolo de comunicación mencionado anteriormente. Primero, el mensaje debe ser codificado usando `codificar_mensaje`. Luego, debes obtener el largo del mensaje en 4 *bytes* y ocupar el `byteorder` *big endian* (`"big"`). Después, debes enviar el largo del mensaje junto al mensaje codificado (en ese orden) al servidor. Finalmente, debes resetear `evento_prints`.

Notas

- 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.
- Eres libre de crear nuevos atributos y métodos que creas necesarios para el desarrollo de tu programa.

Objetivos

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

Anexo

Formato mensaje cliente-servidor

El cliente y el servidor intercambian información a través de mensajes que siguen un formato específico que incluye el comando a ejecutar y los argumentos que lo acompañan. La forma general es la siguiente:

```
{
    "comando": <comando>,
```

```
    "argumentos": {
        <arg 1>: <valor 1>,
        <arg 2>: <valor 2>,
        ...
    }
}
```

Ejemplo: cliente le pide a servidor descargar el archivo ”meme.png”.

Request del cliente:

```
{
    "comando": "descargar",
    "argumentos": {
        "tipo": "foto",
        "nombre": "meme",
    }
}
```

Respuesta del servidor

```
{
    "comando": "archivo",
    "argumentos": {
        "contenido": <bytes archivo>,
        "ruta": <ruta a guardar el archivo>,
    }
}
```

Comandos del servidor

Comandos que recibe el servidor:

- **almacenamiento:** Le retorna al cliente la cantidad de espacio ocupado en el servidor.
- **explorar:** Recorre los archivos que tiene el servidor en `data/` y se los retorna al cliente.
- **explorar_descargar:** Recorre los archivos que tiene el servidor en `data/` y se los retorna al cliente. A pesar de que retorna el mismo contenido que **explorar**, se hace la distinción entre ambos de la forma en que en este caso el cliente sepa que, al recibir este comando como respuesta, debe pedirle al usuario un nuevo input que indique el archivo a descargar.
- **descargar:** Se recibe, junto a este comando, el tipo y nombre del archivo que el cliente desea recibir, y le envía de vuelta la ruta del archivo y, de manera separada, el archivo solicitado por chunks de 8 kb.
- **subir:** Se recibe, junto a este comando, un archivo enviado por el cliente (y las especificaciones de su tipo y nombre con el que se debe guardar), el cual se guarda en su directorio correspondiente dentro del directorio `data/` del servidor.