Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2025-1

Publicación: 22 de Mayo de 2025
Actividad evaluada

Actividad 7
Serialización y Networking I

Entrega

- **Lugar:** Repositorio personal de GitHub — Carpeta: `Actividades/AC7`
- **Fecha máxima de entrega:** 22 de Mayo 17:20
- **Ejecución de actividad**: La Actividad será ejecutada **únicamente** desde la terminal del computador. **Los *paths* relativos utilizados en la Actividad deben ser coherentes con esta instrucción, y no pueden modificarse.**

Introducción

Después de la semana de receso, los estudiantes del Departamento de Ciencia de la Computación (DCC) se han dedicado a salir con tus amigos: restaurantes, cines, karaokes, etc. Cansados de complicarse con la división de las cuentas, te piden que implementes un programa que permita dividir los gastos de estas salidas y revisar las deudas que cada estudiante tiene. En base a esto, decides utilizar tus conocimientos de Serialización y *Networking* para implementar **DCCuentasPendientes**.

Flujo del programa

El programa consiste en dos *scripts* independientes:

- Un **servidor** que permite la conexión de múltiples clientes. Este servidor administra la información de los usuarios, guardando su información en un archivo JSON.
- Un **cliente** que se comunica comunica con el servidor para permitiéndoles consultar sus deudas y agregar nuevas transacciones.

La comunicación entre el clientes y el servidor se realiza por medio de *sockets*, donde los mensajes son serializados mediante el uso de la librería `pickle`.

Esta actividad consta en completar 2 clases utilizando los contenidos de Serialización y *Networking* I. Estas dos clases implementan una arquitectura cliente-servidor, donde una clase modela el cliente, mientras que la otra corresponde al servidor. Ambas clases serán corregidas exclusivamente mediante el uso de *tests*.

Finalmente, debes asegurarte de entregar -como mínimo- los archivos que tenga el *tag* de `Entregar` en la siguiente sección, asegurando que se mantenga la estructura de directorios original de la actividad. Los demás archivos no es necesario subir, pero tampoco se penalizará si se suben al repositorio personal.

Archivos y entidades

En el directorio de la actividad encontrarás los siguientes archivos:

- Entregar Modificar `cliente/main.py`: Archivo principal a ejecutar del cliente. Contiene la definición de la clases `Cliente`, que presenta los siguientes atributos:
  - `self.host` — *String* correspondiente a la dirección IP del servidor.
  - `self.port` — *Integer* correspondiente al puerto del servidor.
  - `self.socket` — Instancia de `socket` que utiliza el Cliente para comunicarse con el Servidor.
  - `self.acciones` — Diccionario de acciones que maneja el cliente.

  Los métodos a completar serán explicados con mayor detalle en la Parte II y Parte III. El resto de los métodos no serán explicados en el enunciado, pero cuentan con un *docstring*[1] que permite comprender para qué sirve cada método.
- Entregar Modificar `servidor/main.py`: Archivo principal a ejecutar del servidor. Contiene la definición de la clase `Servidor`, la cual presenta los siguientes atributos:
  - `self.host` — *String* correspondiente a la dirección IP del servidor.
  - `self.port` — *Integer* correspondiente al puerto del servidor.
  - `self.socket_servidor` — Instancia de `socket` que utiliza el Servidor para recibir conexiones de Clientes.
  - `self.clientes` — Diccionario con la información de los clientes.
  - `self.contador_clientes` — *Integer* que cuenta los clientes.
  - `self.transacciones` — Diccionario con el registro de las tracciones.
  - `self.path_transacciones` — *String* que indica la ubicación del archivo que almacena las transacciones.
  - `self.acciones` — Diccionario de acciones que maneja el servidor.

  Los métodos a completar serán explicados con mayor detalle en la Parte I, Parte II y Parte III. El resto de los métodos no serán explicados en el enunciado, pero cuentan con un *docstring* que permite comprender para qué sirve cada método.
- No modificar `servidor/data.json`: Archivo que almacena los prestamos realizados. Contiene los nombre de los deudores, los cuales están asociados a diccionarios con sus deudas. Estos diccionarios contiene los nombres de sus prestadores y los montos asociados.
- No modificar `utils.py`: Archivo presente tanto en la carpeta `cliente/` como `servidor/`. Presenta la clase `Mensaje`, la cual se utilizará para comunicar el cliente y el servidor. Contiene los siguientes atributos:
  - `self.accion` — *String* que indica la acción asociada a la solicitud del mensaje.
  - `self.argumentos` — Diccionario correspondiente a los argumentos de la solicitud.
  - `self.respuesta` — Objeto correspondiente a la respuesta de la solicitud.

[1] En caso de que quieras aprender más sobre los *docstrings* te recomendamos revisar el siguiente *link*.

Parte I. Cargar y guardar transacciones

Antes de enfocarnos en que la comunicación entre el cliente y el servidor, deberás asegurarte de que el servidor sea capaz de leer las información del archivo `data.json`. Para esto, deberás completar los siguientes métodos de la clase `Servidor` mediante el **correcto uso de JSON**:

- Modificar `def cargar_transacciones(self) -> None`:

  Método encargado de cargar el contenido del archivo de transacciones y almacenar su contenido en el atributo `self.transacciones`.

  Cargar el archivo indicado en `self.path_transacciones`, utilizando *encoding* UTF-8.

- Modificar `def guardar_transacciones(self) -> None`:

  Método encargado de guardar las transacciones indicadas en el atributo `self.transacciones`.

  Sobrescribe el archivo indicado en `self.path_transacciones`, utilizando *encoding* UTF-8. Se recomienda utilizar el argumento `indent=4`, para facilitar la lectura del archivo.

Parte II. Conexión entre cliente y servidor

Dado que ya hemos asegurado que el servidor es capaz de cargar y guardar la información de las transacciones, ahora nos enfocaremos en la conexión entre el cliente y servidor. Deberás completar los siguientes métodos mediante el **correcto uso de *sockets***:

Clase `Cliente`:

- Modificar `def __init__(self, host: str, port: int) -> None`:

  Inicializador de la clase. Asigna los argumentos recibidos en los atributos que le corresponden.

  Debes modificar el atributo `self.socket` para asegurar que contenga un *socket*. Además, este *sockets* debe utilizar direcciones IP de tipo IPv4 y el protocolo TCP.

- Modificar `def conectar(self) -> None`:

  Método encargado de conectar el cliente al servidor. Utiliza los datos almacenados en los atributos `self.host` y `self.port`.

Clase `Servidor`:

- Modificar `def __init__(self, host: str, port: int) -> None`:

  Inicializador de la clase. Asigna los argumentos recibidos en los atributos que le corresponden.

  Debes modificar el atributo `self.socket_servidor` para asegurar que contenga un *socket*. Además, este *sockets* debe utilizar direcciones IP de tipo IPv4 y el protocolo TCP.

- Modificar `def bind_listen(self) -> None`:

  Método encargado de asociar el *socket* el servidor a la IP y puerto entregados. Además, habilita el servidor para que pueda recibir conexiones.

- Modificar `def aceptar_clientes(self) -> None:`

  Método encargado de aceptar las conexiones de los clientes. Una vez aceptada la solicitud, le asigna un id al cliente y se almacena su información en el diccionario `self.clientes`, asegurando que se mantenga la siguiente estructura:

```
{
    id_cliente: (socket_cliente, dirección_cliente)
}
```

- Modificar `def desconectar_cliente(self, id_cliente: int) -> None:`

  Método encargado de manejar la desconexión de un cliente. Cierra el *socket* asociado al cliente y elimina su entrada del diccionario `self.clientes`.

Parte III. Manejo de mensajes

Finalmente, dado que ahora es posible conectar el cliente y el servidor, nos enfocaremos en la comunicación entre ambos. Utilizando `pickle` deberás completar los siguientes métodos:

Clase `Cliente`:

- Modificar `def enviar_mensaje(self, mensaje: Mensaje) -> bytes:`

  Recibe una instancia de `Mensaje` y lo serializa a *bytes* mediante el uso de la `pickle`. Una vez serializado, lo envía al servidor.

  Para facilitar el testeo de este método y poder validar que los datos se están serializando correctamente, debes retornar los *bytes* obtenidos al serializar el mensaje.

- Modificar `def recibir_mensaje(self) -> Mensaje:`

  Recibe los *bytes* enviados por el servidor y los deserializa obteniendo así una instancia de `Mensaje`. Retorna el mensaje obtenido.

  Para simplificar las recepción de mensajes, puedes asumir que ningún mensaje tendrá más de 8000 *bytes*.

Clase `Servidor`:

- Modificar `def enviar_mensaje(self, id_cliente: int, mensaje: Mensaje) -> bytes:`

  Recibe el id de un cliente y una instancia de `Mensaje`. Serializa el mensaje a *bytes* mediante el uso de la `pickle` y lo envía al cliente que corresponde.

  Para facilitar el testeo de este método y poder validar que los datos se están serializando correctamente, debes retornar los *bytes* obtenidos al serializar el mensaje.

- Modificar `def recibir_mensaje(self, id_cliente: int) -> Mensaje:`

  A partir del id entregado, recibe los *bytes* enviados por el cliente y los deserializa obteniendo así una instancia de `Mensaje`. Retorna el mensaje obtenido.

  Para simplificar las recepción de mensajes, puedes asumir que ningún mensaje tendrá más de 8000 *bytes*.

Notas

- **No puedes hacer *import* de otras librerías externas a las entregadas en el archivo.**
- Recuerda que la ubicación de tu entrega es en **tu repositorio de Git**. En la rama (*branch*) por defecto del repositorio: `main`.
- Se recomienda completar la actividad en el orden del enunciado.
- Recuerda que esta evaluación presenta corrección **automatizada**. Si entregas un código que se cae al momento de correr los *tests*, será evaluado con 0 puntos.
- Si aparece un error inesperado, ¡léelo y revisa el código del *test*! Intenta interpretarlo y/o buscarlo en Google.
- Se recomienda probar tu código con los *tests* y ejecutando `main.py`, este último se ofrece un pequeño código donde se prueba la carrera con 3 jugadores.

Objetivo de la actividad

- Aplicar conocimientos de Serialización mediante el uso de las librerías `pickle` y `json`.
- Aplicar conceptos de la arquitectura cliente-servidor.
- Utilizar *sockets* para comunicar un cliente y servidor.
- Probar código mediante la ejecución de *test* y de los archivos `main.py`.

Ejecución de código

Por lo general -en este curso- cuando trabajamos en una arquitectura cliente-servidor, se separamos el código del cliente y servidor en carpetas independientes, para así simular computadores independientes. Por lo que, para ejecutar el código, se ubica la terminal en cada una de las carpetas y se ejecuta el código.

Dado que en el caso de esta Actividad utilizamos tests para ejecutar el código, para ejecutar el código del servidor y el cliente debes ubicarte `Actividades/AC7` y ejecutar los siguientes comandos:

- `python3 servidor/main.py`
- `python3 cliente/main.py`

Importante: recuerda que si `python3` no funciona, probar con el comando específico de tu computador. Este puede ser `py`, `python`, `py3` o `python3.11`.

Ejecución de *tests*

En esta actividad se provee de varios archivos `.py` los cuáles contiene diferentes *tests* que ayudan a validar el desarrollo de la actividad. Para ejecutar estos *tests*, **primero debes posicionar tu terminal/consola en la carpeta de la actividad (`Actividades/AC7`)**. Luego, desde esta misma, debes escribir el siguiente comando para ejecutar todos los *tests* de la actividad:

- `python3 -m unittest discover tests_publicos -v -b`

En cambio, si deseas ejecutar un subconjunto de *tests*, puedes hacerlo si escribes lo siguiente en la terminal/consola:

- `python3 -m unittest -v -b tests_publicos.test_servidor_cargar_datos`
  Para ejecutar solo el subconjunto de *tests* de la Parte I.
- `python3 -m unittest -v -b tests_publicos.test_cliente_conexion`
  Para ejecutar solo el subconjunto de *tests* de la Parte II del Cliente.
- `python3 -m unittest -v -b tests_publicos.test_servidor_conexion`
  Para ejecutar solo el subconjunto de *tests* de la Parte II del Servidor.
- `python3 -m unittest -v -b tests_publicos.test_cliente_mensajes`
  Para ejecutar solo el subconjunto de *tests* de la Parte III del Cliente.
- `python3 -m unittest -v -b tests_publicos.test_servidor_mensajes`
  Para ejecutar solo el subconjunto de *tests* de la Parte III del Servidor.

Importante: recuerda que si `python3` no funciona, probar con el comando específico de tu computador. Este puede ser `py`, `python`, `py3` o `python3.11`.