Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2025-2

Publicación: 30 de octubre de 2025
Actividad evaluada

Actividad 6
Serialización y *Networking* II

Entrega

- **Lugar:** Repositorio personal de GitHub — Carpeta: `Actividades/AC6`
- **Fecha máxima de entrega:** 30 de octubre 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.**

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**.

Objetivo de la actividad

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

Introducción

Después de días utilizando *DCCitas* y sin tu alma gemela a la deriva, decides utilizar otros estrategias. Pensando en lo mucho que te gustan las mascotas, decides utilizar tus nuevos conocimientos de Serialización y *Networking* para implementar una aplicación que permita compartir fotos de mascotas y así encontrar a tu alma gemela.

Flujo del programa

El programa consiste en dos *scripts* independientes:

- Un **servidor** que permite la conexión de múltiples clientes. Este servidor administra distintos archivos que los clientes pueden recibir.
- Un **cliente** que se comunica con el servidor para consultar los archivos que almacena y pedirle que le envié dichos archivos.

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* II. 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 y carpetas:

- Entregar Modificar `cliente/main.py`: Archivo que contiene la definición de la clase `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.archivo_nombre` — *String* que indica el nombre del archivo que se está pidiendo.
  - `self.archivo_completo` — *Booleano* que indica si actualmente se está pidiendo un archivo completo o no.
  - `self.chunks_totales` — *Integer* correspondiente a la cantidad de *chunks* que se utilizarán para enviar el archivo.
  - `self.chunk_actual` — *Integer* correspondiente a el *chunks* actual del archivo.
  - `self.bytes_por_chunk` — Diccionario en el que se guardan todos los *bytes* recibidos. Como llave se utiliza el *chunk* y como valor los *bytes* asociados.
  - `self.data_path` — *String* correspondiente a la carpeta donde se guardaran los archivos recibidos.
  - `self.acciones` — Diccionario de acciones que maneja el cliente.

  Los métodos a completar serán explicados con mayor detalle en la Parte I y Parte II. 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.

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

- Entregar Modificar `servidor/main.py`: Archivo que 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.tamano_chunk` *Integer* correspondiente al tamaño de los *chunks* a enviar.
  - `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.solicitudes_archivos` Diccionario con el registro de los archivos que está solicitando cada cliente.
  - `self.data_path` *String* correspondiente a la carpeta que contiene todos los archivos que almacena el servidor.
  - `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/`: Carpeta que almacena los distintos archivos que almacena el servidor.
- 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.
- No modificar `servidor.py`: Archivo principal para ejecutar el servidor.
- No modificar `cliente.py`: Archivo principal para ejecutar el cliente.

Parte I. Conexión entre cliente y servidor

Antes de preocuparnos por el envío de archivos, 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 II. Manejo de mensajes

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*.

Parte III. Transferencia de archivos

Finalmente, para que esta aplicación cumpla su funcionalidad, se debe implementar el envío y la recepción de archivos. Para asegurar lo anterior, se utilizará el siguiente protocolo:

1. Cliente solicita un archivo (`Cliente.registrar_solicitud_archivo()`). Servidor registra dicha solicitud (`Servidor.solicitar_archivo()`) y retorna la cantidad de *chunks* que serán necesarios para enviar el archivo completo.
2. Cliente empieza a pedir el archivo completo (`Cliente.pedir_archivo_completo()`). Para esto, actualiza el valor de los atributos `archivo_completo` y `bytes_por_chunk`, y empieza a pedir al Servidor cada unos de los *chunks* del archivo.

   Mientras, no se hayan pedido todos los *chunks* del archivo:

   2.1. Cliente pide al Servidor el *chunk* que le corresponde actualmente (`Cliente.pedir_chunk()`).

   2.2. Servidor recibe la solicitud del *chunk* y resonde con los *bytes* correspondientes (`Servidor.solicitar_chunk()`).

   2.3. Cliente recibe los *bytes* y los guarda en el diccionario `bytes_por_chunk` (`Cliente.guardar_chunk()`).

   2.4. Una vez guardado los *bytes*, se aumenta el valor del *chunk* actual.
3. Finalmente, cuando se hayan recibido todos los *chunks* asociados al archivo:

   3.1. Cliente junta todos los *chunks* y se escribe el archivo correspondiente (`Cliente.guardar_archivo()`).

   3.2. Cliente restablece los atributos asociados a la solicitud de un archivo y envía al Servidor un mensaje indicado que terminó la solicitud del archivo (`Cliente.terminar_solicitud_archivo()`).

   3.3. Servidor termina la solicitud del archivo (`Servidor.terminar_solicitud_archivo()`).

En base a lo anterior, deberás completar los siguientes métodos:

Clase `Servidor`:

- Modificar `def solicitar_chunk(self, id_cliente: int, n_chunk: int) -> bytes:`

  Método encargado de leer el archivo asociado al cliente y retorna los *bytes* correspondiente al *chunk* indicado (`n_chunk`).

Clase `Cliente`:

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

  Método encargado de concatenar todos los archivos almacenados en el diccionario `bytes_por_chunk` y guardarlos en un archivo en el lado del cliente.

  Es importante tener en consideración que:

  - Los *bytes* deben concatenarse según el *chunk* asociado a cada grupo de *bytes* de forma ascendente.
  - El archivo debe guardarse en el directorio indicado en la `self.data_path`. Además, el nombre y la extensión deben ser los guardados en `self.archivo_nombre`.

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

  Método encargado de restablecer los valores por defecto de los atributos asociados a la solicitud de un archivo: `archivo_completo`, `archivo_nombre`, `chunks_totales`, `chunk_actual` y `bytes_por_chunk`.

  Adicionalmente, **envía un mensaje al Servidor** indicando que se completó la solicitud. La acción asociada a dicho mensaje debe ser `'terminar_solicitud_archivo'` y no incluye argumentos.

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.

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/AC6` y ejecutar los siguientes comandos:

- `python3 servidor.py XXXX`
- `python3 cliente.py XXXX`

donde `XXXX` corresponde a un puerto. Si no se entrega, se utilizará un valor por defecto.

Importante: recuerda que si `python3` no funciona, probar con el comando específico de tu computador. Este puede ser `py`, `python`, `py3` o `python3.12`.

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/AC6`)**. 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_cliente_conexion`
  Para ejecutar solo el subconjunto de *tests* de la Parte I del Cliente.
- `python3 -m unittest -v -b tests_publicos.test_servidor_conexion`
  Para ejecutar solo el subconjunto de *tests* de la Parte I del Servidor.
- `python3 -m unittest -v -b tests_publicos.test_cliente_mensajes`
  Para ejecutar solo el subconjunto de *tests* de la Parte II del Cliente.
- `python3 -m unittest -v -b tests_publicos.test_servidor_mensajes`
  Para ejecutar solo el subconjunto de *tests* de la Parte II del Servidor.
- `python3 -m unittest -v -b tests_publicos.test_transferir_archivos`
  Para ejecutar solo el subconjunto de *tests* de la Parte III del Servidor y Cliente.

Importante: recuerda que si `python3` no funciona, probar con el comando específico de tu computador. Este puede ser `py`, `python`, `py3` o `python3.12`.