Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2019-2

27 y 28 de noviembre de 2019
Actividad Sumativa

Actividad 10

Networking

Introducción

Despues de subir la Tarea 03, el malvado Dr.Pinto descubrió la última creación del antiguo Dr. Valdivieso: el `Enz1-nat0r`, cuya misión es eliminar a todos los alumnos de Programación Avanzada.

Para detener a Dr. Pinto, los ayudantes del curso rápidamente hackearon al `Enz1-nat0r` para descubrir su única debilidad: el juego **DCCuatro en Línea**. Los ayudantes lograron avanzar un poco en el desarrollo del juego, dejando la lógica ya implementada. Sin embargo, para poder luchar contra el `Enz1-nat0r` usando este juego, es necesario que pueda ser utilizado a través de *networking* y es tu misión completarlo.

Figura 1: Imagen de `Enz1-nat0r`

DCCivil War: DCCuatro en Línea

El DCCuatro en Línea es un juego en el que dos jugadores toman turnos para rellenar un tablero vertical con fichas de un color para cada jugador. En cada turno, el jugador decide en qué columna introducir su ficha y esta queda en la posición desocupada de más abajo en esa columna.

El objetivo de los jugadores es ser el primero en lograr colocar cuatro fichas en línea continua en dirección vertical, horizontal o diagonal. En el caso de que todas las casillas del tablero están ocupadas y ningún jugador cumple la condición para ganar, se declara un empate.

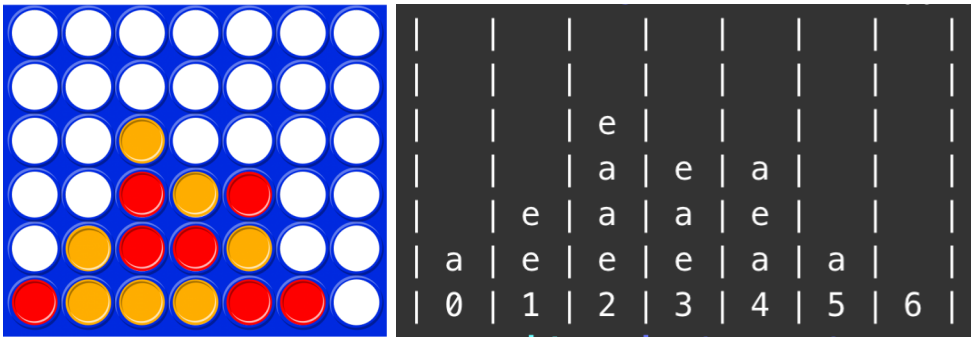

Figura 2: Ejemplo de tablero DCCuatro en Línea en consola. Las fichas del usuario se marcan con una `a`, mientras que las del servidor con una `e`.

Flujo de interacción

El usuario de DCCuatro en Línea ejecutará un cliente para jugar contra el servidor. El servidor es el primero en ser ejecutado y esperará a un cliente que desee conectarse. **Solo interactúa con un cliente a la vez** y no cambia de cliente hasta que el actual cierre la conexión. Una vez conectados, el servidor queda escuchando la llegada de **acciones**, y por cada posible acción genera un efecto en el estado del juego o en la comunicación. El cliente recibe como *input* **comandos del usuario** a través de la consola, y de acuerdo a ellos envía acciones correspondientes al servidor.

Los **tres comandos** que se le ofrecen al usuario y que debe ingresar en consola (incluyendo \) son:

- `\juego_nuevo`: asociado a la acción que reinicia el juego actual, o inicia uno si no había antes. El juego no comienza hasta que se ha enviada esta acción, y el servidor responde con el estado del tablero nuevo.
- `\jugada columna`: asociado a la acción de realizar una jugada en el tablero actual, donde `columna` es un entero que especifica la columna donde el usuario desea poner su ficha. El cliente solo verificará que el número sea efectivamente un entero, pero es la lógica del juego la que verifica que el número ingresado sea una jugada válida. Si el juego está iniciado, el servidor responde con el estado actual del tablero tras la jugada. De no haber iniciado un juego aún o no es una jugada válida, responde diciendo que es un comando inválido.
- `\salir`: asociado a la acción de terminar de interactuar y desconectar al cliente del servidor.

Una vez finalizado el juego (alguien gana o hay un empate), el juego se acaba y se espera a que el usuario ingrese un comando que lo reinicie para volver a jugar, o se desconecte.

La gran mayoría del flujo anterior se encuentra implementado en las clases `Cliente` y `Servidor`, respectivamente en las carpetas `cliente` y `servidor`. En estas clases **se marcan las secciones que debes completar o modificar**, y corresponden a las secciones de comunicación entre programas. La clase `Juego` se entrega **completamente implementada** y encapsula la lógica del juego. Además, es importada por `Servidor` y su uso en él también se entrega implementado.

A continuación se detallan los métodos a completar de los módulos entregados. La Figura 3 muestra un diagrama de flujo con el orden de ejecución de los distintos métodos que interactúan a través de componentes. **Te recomendamos fuertemente revisar este diagrama a medida que lees e implementas los métodos pedidos.** Así podrás entender mejor el flujo y establecer en que orden implementar cada uno de los métodos. Por ejemplo, es buena idea implementar un método de envío de datos **previo** a implementar el método que reciba los datos enviados.

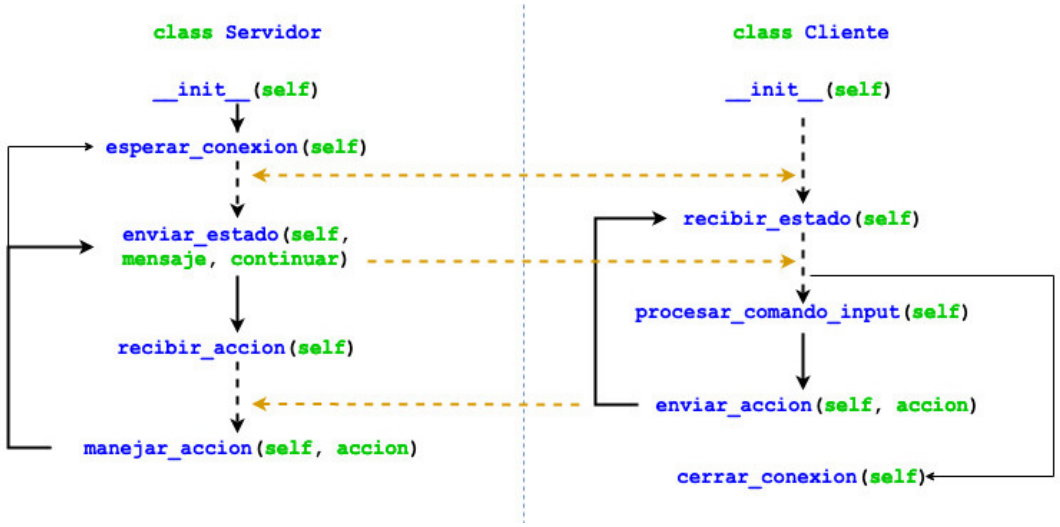

Figura 3: Flujo de ejecución. Las flechas indican el orden en que se ejecutan métodos, comenzando por los `__init__`. Las flechas punteadas bloquean la ejecución en espera de interacción mediante *sockets*, y las amarillas indican el momento de estas interacciones a través de *sockets*.

Servidor: `servidor/main.py`

El servidor es responsable de: enviar el estado del juego al cliente, recibir las acciones enviadas por el cliente y manejar la lógica del juego. Esta última se se encuentra totalmente implementada, por lo que no es necesario modificarse. Los siguientes métodos de `class Servidor` son aquellos que debes completar para lograr la comunicación de este módulo con `Cliente`:

- `def __init__(self)`: este método se encarga de inicializar el servidor, creando su *socket* capaz de escuchar conexiones e inicializar el juego vacío. Se entrega parcialmente implementado, donde debes completar los detalles de especificación de *host* y *port*, creación del *socket* del servidor y habilitarlo para escuchar conexiones.
- `def esperar_conexion(self)`: este método se encarga de esperar y aceptar la conexión directa de un cliente, y así obtener su respectivo *socket*. Debes completar esos detalles de conexión. Una vez efectuada la conexión, se ejecuta el método `interactuar_con_servidor` que se entrega implementado y es el ciclo principal del servidor para interactuar con el cliente.
- `def enviar_estado(self, mensaje, continuar)`: este método recibe un *string* `mensaje` y un *boolean* `continuar`. Tiene por objetivo enviar el estado actual del juego al cliente. Lo hace concatenando a `mensaje` el estado del tablero y los comandos que puede realizar el usuario, y mediante `continuar` se indica si el cliente debe seguir conectado o no. Debes realizar el envío a través de *sockets* de esta información y de tal forma que funcione para cualquier largo de información. La manera en que envíes los correspondientes *bytes* debe poder ser decodificada por el método `recibir_estado` que debes implementar en `Cliente`.
- `def recibir_accion(self)`: este método recibe desde el cliente el mensaje que contiene la acción asociada al comando realizado por el usuario, y que fue codificado por el método `enviar_accion` de `Cliente`. El método `recibir_accion` se encuentra parcialmente implementado, así que debes modificar los detalles de recepción de *bytes* y debes generalizar el protocolo para poder recibir cualquier cantidad de *bytes*.

- `def manejar_accion(self, accion):` este método recibe la acción enviada por el cliente ya decodificada, y ejecuta los distintos tipos de efectos dependiendo de la acción recibida. Se encuentra implementado casi en su totalidad, solo debes completar las secciones marcadas que extraen el tipo de acción y los datos de jugada que provengan en `accion`. Recuerda que desde `Cliente` decidirás cómo se envían estos datos.

Cliente: `cliente/main.py`

El cliente es el encargado de: recibir el estado del juego desde el servidor, obtener los comandos del usuario y enviarla la acción asociada al servidor. Para lograrlo, los siguientes métodos de `class Cliente` son los que debes completar:

- `def __init__(self):` el constructor de la clase que se encarga de realizar la conexión del cliente con un servidor en ejecución. Para esto debes crear los *sockets* correspondientes e intentar establecer la conexión, además de completar los detalles de especificación de *host* y *port*. Una vez lograda la conexión, este método llama a `interactuar_con_servidor` que se entrega implementado y es el ciclo principal del cliente para interactuar con el servidor.
- `def recibir_estado(self):` método destinado a recibir una actualización del estado del juego desde el servidor, codificado por el método `enviar_estado` en `Servidor`. Debe retornar un *string* con el mensaje a imprimir en consola para el usuario, y un *boolean* que especifique si debe continuar funcionando. Este método se encuentra parcialmente implementado, ya que solo recibe un número fijo de *bytes*. Debes modificarla para que pueda recibir una cantidad indefinida de *bytes*, a través de un protocolo generalizado de tu elección.
- `def procesar_comando_input(self):` método que pide el *input* del usuario como *string* y revisa que corresponda a uno de los tres comandos válidos listados anteriormente. Si no corresponde a ninguno, debe retornar `None`. Pero si efectivamente corresponde a uno, debe retornar un objeto de tu preferencia que contenga la información de la acción asociada para que sea luego enviado al servidor. Para el caso de `\jugada columna`, se espera también que se verifique aquí que `columna` sea un número entero[1], **pero no debe verificar que sea una jugada válida en el tablero**.
- `def enviar_accion(self, accion):` este método se encarga de enviar la acción con la información del comando ingresado por el usuario al servidor y que será luego recibido e interpretado por el método `recibir_accion` en `Servidor`. Queda a tu criterio la forma en la que estructuras el objeto procesado y luego enviado, pero debe ser consistente con lectura que se encuentra implementada en el servidor. También, debes tener en consideración que pueda enviar una cantidad indefinida de *bytes* para ser recibida por el servidor.

Bonus

Se asignará una bonificación de **5 décimas** en nota si además de implementar los métodos de envío y recepción de información en `Servidor` y `Cliente`, se utilizan:

- `pickle` para serializar la información enviada desde `Servidor` hacia `Cliente`.
- `json` para serializar la información enviada desde `Cliente` hacia `Servidor`.

[1] Para revisar si un *string* específico corresponde a un entero, puedes usar el método `str.isnumeric()`

Notas

- Te recomendamos leer **detalladamente** el código que se encuentra implementado, esto de dará una mejor idea de como está estructurado y como debes completarlo.
- Atrévete a colocar `print`s en varias partes del código para entender como funciona.
- También es recomendable seguir el orden de métodos según el flujo indicado en la Figura 3.
- El uso de diccionarios para enviar los mensajes puede ser de gran utilidad.
- Solo es necesario entregues los correspondientes archivos `.py` de cada módulo.
- Recuerda, *habemus bonus*

Requerimientos

- (3.25 pts) `class Servidor`
  - (1.00 pt) `def __init__`:
    - (0.25 pts) Se usan los argumentos correctos para *host* y *port*.
    - (0.50 pts) Se inicia el *socket* correctamente.
    - (0.25 pts) El *socket* escucha conexiones
  - (0.50 pts) `def esperar_conexion`:
    - Se conecta y guarda el *socket* del cliente adecuadamente.
  - (0.50 pts) `def enviar_estado`:
    - (0.25 pts) Se serializa el mensaje.
    - (0.25 pts) Se envía el mensaje serializado adecuadamente.
  - (1.00 pt) `def recibir_accion`:
    - (0.75 pts) Se recibe adecuadamente el mensaje serializado.
    - (0.25 pts) Se deserializa.
  - (0.25 pts) `def manejar_accion`:
    - Se obtienen los comandos del cliente.
- (2.75 pts) `class Cliente`
  - (0.75 pts) `def __init__`:
    - (0.25 pts) Se usan los argumentos correctos para *host* y *port*.
    - (0.50 pts) Se inicia el *socket* correctamente.
  - (1.00 pt) `def recibir_estado`:
    - (0.75 pts) Se recibe adecuadamente el mensaje serializado.
    - (0.25 pts) Se deserializa.
  - (0.50 pts) `def procesar_comando_input`:

    - Se revisa si el *input* del usuario es valido y se retorna correctamente en ese caso.
- (0.50 pts) `def enviar_accion`:
    - (0.25 pts) Se serializa el mensaje.
    - (0.25 pts) Se envía el mensaje serializado adecuadamente.

Bonus

- (0.50 pts) *Bonus*
    - (0.25 pts) Se envía la informacion del Servidor al Cliente usando `pickle`.
    - (0.25 pts) Se envía la informacion del Cliente al Servidor usando `json`.

Entrega

- **Lugar:** En su repositorio privado de GitHub, en la **carpeta** `Actividades/AC10/`
- **Hora del *push*:** 20:00, 28 de noviembre