Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2020-2

19 de Noviembre de 2020
Actividad Formativa

Actividad Formativa 05

Networking

Entrega

- **Lugar:** En su repositorio privado de GitHub, en la **carpeta** `Actividades/AF05/`
- **Hora del *push*:** 16:50

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

Hemos llegado al final del semestre, has pasado por incontables desafíos que te han permitido llegar hasta aquí. Gracias a tu destacado talento programando, necesitamos que le des una última mano al DCC.

En la nave del DCCrew se han infiltrado impostores que quieren destruir el DCC, eliminando a todos los tripulantes de la nave.

Para evitar que esto suceda y como toda decisión en la nave del DCCrew es DemoCratiCa deberás implementar un sistema para que entre todos podamos conversar y votar a quien abandonará la nave **¡para siempre!** y así desenmascarar a los impostores. Sólo tú y tus conocimientos sobre *networking* podrán salvarnos.

Flujo del Programa

DemoCratiCa es un programa en red que se compone de dos partes: el **Cliente** y el **Servidor**. Ambas partes deben ejecutarse en procesos diferentes de manera simultánea, ya sea en computadores diferentes o en el mismo, y se envían mensajes entre sí usando la dirección IP y el puerto de cada uno. Para esta actividad debes completar estos programas de manera que se pueda establecer una conexión, y usar un protocolo para comunicar cliente con servidor, y permitir que múltiples clientes puedan conectarse con el mismo servidor, de manera que varios usuarios puedan interactuar en el juego. Podrás ver parte del flujo a través de los **logs** (o mensajes de consola) que serán impresos con valiosa información, a medida que trabaje el servidor.

Podrás trabajar con **uno o más** clientes propios (otros serán *bots* generados automágicamente) durante la implementación del programa. Los clientes ya están incorporados por lo que sólo te preocuparás de la **conexión cliente-servidor**. La sección Ventanas muestra una visualización del flujo general del juego con dos clientes.

En la carpeta de **AF05** se te entrega una carpeta para el cliente y una para el servidor. Cada carpeta contiene los archivos necesarios para su ejecución. Los roles y funcionalidades, además del contenido del directorio con el cual trabajarás en esta actividad se detallan a continuación:

Directorio Actividad

```
├── servidor/
│   ├── jugadores.py
│   ├── logica.py
│   ├── main.py
│   ├── nombres.txt
│   └── servidor.py ## <--- Parte I
└── cliente/
    ├── assets/
    ├── chat_widget.py
    ├── cliente.py ## <--- Parte II
    ├── interfaz.py
    ├── main.py
    ├── popups.py
    ├── sala_de_espera.py
    ├── ventana_inicio.py
    └── ventana_principal.py
```

Servidor

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

- `nombres.txt`: Archivo de texto, contiene los nombres por defecto que usarán los bots.
- `jugadores.py`: Contiene la clase `Jugador`, que le permite al servidor guardar la información de los jugadores de la partida y de los clientes que se conecten, para poder guardar sus atributos y comunicarse con ellos [**NO debes modificar este archivo**]. La clase `Jugador` contiene el atributo `socket_cliente` el cual puede tomar dos valores:
  - `socket`: El `Jugador` es un cliente real conectado desde un proceso y manejado por una persona.

- `None`: El `Jugador` es creado por el servidor (¡un *bot*!), por lo que no es manejado por una persona. Sus decisiones son generadas por el servidor, lo cual ya se encuentra implementado.

- `logica.py`: Contiene a la clase `Logica`, que implementa la lógica y el flujo del juego. Posee el método `manejar_mensaje(self, mensaje, cliente)`, que recibe un mensaje y el cliente que lo envió y en base a esto genera una respuesta que luego envía a los clientes mediante los métodos `enviar` y `enviar_a_todos`. [**NO debes modificar este archivo**]

  IMPORTANTE: El método `manejar_mensaje()`, **deberás utilizarlo en la Parte II**, por lo que es muy importante que lo revises antes de comenzar a trabajar en esa parte.

- `main.py`: Archivo principal del servidor. Instancia la clase `Servidor` e inicia su funcionamiento para aceptar conexiones de clientes. [**NO debes modificar este archivo**]

- `servidor.py`: Es el módulo que contiene el funcionamiento del servidor. En él encontrarás la clase `Servidor` que contiene los atributos y métodos necesarios para la conexión y comunicación con los clientes. **Debes completar esta clase en la Parte I**.

Parte I (`servidor.py`)

Tu trabajo consiste en completar la clase `Servidor` del archivo `servidor.py`. Los siguientes métodos de la clase `Servidor` **ya están implementados y NO debes modificarlos**:

- `def __init__(self, host, port, log_activado=True)`: Inicializa el servidor, creando un *socket* capaz de escuchar a usuarios. Atributos importantes:
  - `self.log_activado`: Es un `bool` que indica si está permitido imprimir mensajes en la consola.
  - `self.socket_server`: Es el `socket` del servidor, desde el cual se deben aceptar conexiones.
  - `self.lista_jugadores`: Es una lista con instancias de `Jugador` que toman parte del juego. Incluye *bots* y jugadores reales.
  - `self.logica`: Instancia de la clase `Logica`. Es el *backend* del servidor y posee métodos y atributos que describen el flujo del juego.
- `def log(self, mensaje_consola)`: Imprime mensajes en la consola, si la funcionalidad está activada.
- `def eliminar_cliente(self, cliente)`: Elimina un cliente que se encuentre actualmente conectado al servidor. El puesto que este cliente ocupaba se convierte en un *bot*.
- `def decodificar_mensaje(bytes_mensaje)`: Decodifica y deserializa un mensaje usando JSON.
- `def codificar_mensaje(mensaje)`: Codifica y serializa un mensaje usando JSON.
- `def enviar_lista_respuesta(self, jugador, lista_respuestas)`: Recibe una lista con las respuestas, determina a quién enviárselas según su rol y utiliza el método `enviar` para hacer envío de la respuesta correspondiente a cada integrante.
- `def enviar_a_todos(self, mensaje)`: Recibe un mensaje y lo envia a cada cliente conectado al servidor, usando el método `enviar`.

Los métodos que se describirán a continuación **DEBEN ser implementados:**

- `def aceptar_clientes(self)`: Este método se encarga de aceptar **constantemente** conexiones de clientes. Debes utilizar el *socket* del servidor para aceptar **un** cliente y su *socket* respectivo, además debes almacenar la información del usuario recién conectado como una instancia de la clase `Jugador`

y comenzar a recibir información de inmediato, utilizando el método `escuchar_cliente`. Recuerda debes escuchar y conectar a múltiples clientes simultáneamente, por lo que debes implementar una forma en que esto ocurra de manera paralela.

- `def escuchar_cliente(self, jugador):` Este método se encarga de "escuchar" (recibir) continuamente los mensajes enviados por los clientes. Recibe un objeto de la clase `Jugador`, que representa al cliente que está siendo escuchado (recuerda que su socket correspondiente está almacenado en su atributo `socket_cliente`). Luego de recibir cada mensaje, el servidor debe encargarse de procesarlo y generar una respuesta adecuada (ver método `manejar_mensaje` de `Logica`), finalmente debe enviar la respuesta, con ayuda del método `enviar_lista_respuestas`.
- `def enviar(self, mensaje, socket_cliente):` Recibe el `socket` de un cliente y un mensaje, el cual debe ser codificado haciendo uso del método `codificar_mensaje`, luego debe obtener el largo de este mensaje en 5 *bytes* y serializar en *little endian*, finalmente debes hacer envio de este valor y su respectivo mensaje al cliente.
- `def recibir(self, socket_cliente):` Este método recibe los mensajes enviados por un cliente. Lo debes implementar siguiendo estos pasos:
  1. Recibir el largo del mensaje en 5 bytes, el cual contiene `int` serializado en *little endian*.
  2. Luego debes recibir la información en chunks de máximo 64 `bytes` cada uno.
  3. Decodificar el mensaje usando el método `decodificar_mensaje`, y retornarlo.

Cliente

El cliente está contenido en la carpeta `cliente/`, y se compone de los siguientes archivos y directorios:

- `assets/`: Contiene los elementos visuales (imágenes) de la interfaz gráfica del cliente.
- `chat_widget.py`: Crea el *widget* correspondiente al chat y genera sus conexiones. [**NO debes modificar este archivo**].
- `interfaz.py`: Contiene la lógica del *frontend* del cliente, mediante la clase `Controlador(QObject)`. [**NO debes modificar este archivo**].
- `popups.py`: Contiene las clases `PopupCrewmate(QWidget)`, `PopupExpulsar(QWidget)` y `PopupFinal (QWidget)`, las cuales corresponden a los *popups* mostrados a lo largo del programa al ocurrir ciertas acciones. [**NO debes modificar este archivo**].
- `sala_de_espera.py`: Contiene la clase `SalaDeEspera(QMainWindow)`, corresponde a la ventana de la sala de espera del programa. [**NO debes modificar este archivo**].
- `ventana_de_inicio.py`: Contiene la clase `VentanaInicio(QMainWindow)`, corresponde a la ventana de inicio del programa. [**NO debes modificar este archivo**].
- `ventana_de_inicio.py`: Contiene la clase `VentanaPrincipal(QMainWindow)`, corresponde a la ventana principal, contecta todas las ventanas antes mencionadas y está encargada del flujo principal del programa. [**NO debes modificar este archivo**].
- `cliente.py`: Es el módulo que contiene la lógica principal del funcionamiento del cliente. En él encontrarás la clase `Cliente`, la cual contiene los atributos y métodos necesarios para la conexión y comunicación con el servidor. **Debes completar esta clase en la Parte II**.

Parte II (`cliente.py`)

Deberás completar los métodos existentes dentro de la clase `Cliente` del archivo `cliente.py`, a continuación se detallan los métodos que **ya están implementados y NO debes modificar**:

- `def __init__(self, host, port):` Inicializa el cliente, crea un *socket* que será el encargado de comunicarse con el servidor. Los atributos se detallan a continuación:
  - `self.host:` Dirección IP del servidor a conectarse.
  - `self.port:` Puerto del servidor a conectarse.
  - `self.controlador:` Instancia de `Controlador`, que controla la interfaz gráfica.
  - `self.cliente_socket:` El *socket* del cliente, a través del cual se va a comunicar con el servidor.

Los métodos que se detallan a continuación **DEBEN ser implementados:**

- `def enviar(self, mensaje):` Recibe un diccionario y está encargado de serializar, codificar (revisar método `codificar_mensaje()`) y enviar el mismo diccionario al servidor. Deberás señalar el largo del mensaje en los primeros **5** `bytes` en *little endian* para que el servidor pueda manejarlo desde el otro lado.
- `def recibir(self):` Se deben seguir los mismos pasos que en el método `recibir()` del servidor:
  1. Recibir el largo del mensaje en 5 bytes, el cual contiene `int` serializado en *little endian*.
  2. Luego debes recibir la información en chunks de máximo 64 `bytes` cada uno.
  3. Decodificar el mensaje usando el método `decodificar_mensaje`, y retornarlo.
- `def escuchar_servidor(self):` Método encargado de escuchar los mensajes enviados por el servidor mientras se esté conectado. Deberás completar el código faltante dentro del `try/except` para permitir que el cliente escuche el mensaje a través del método `self.recibir()`, y luego entregar el mensaje a la interfaz a través del método `manejar_mensaje()`, del atributo `self.controlador`. Finalmente, deberás cerrar la conexión.

Ventanas

A continuación se muestran las ventanas representativas del flujo general del juego, ordenadas según su aparición:

1. **Ventana de Inicio**

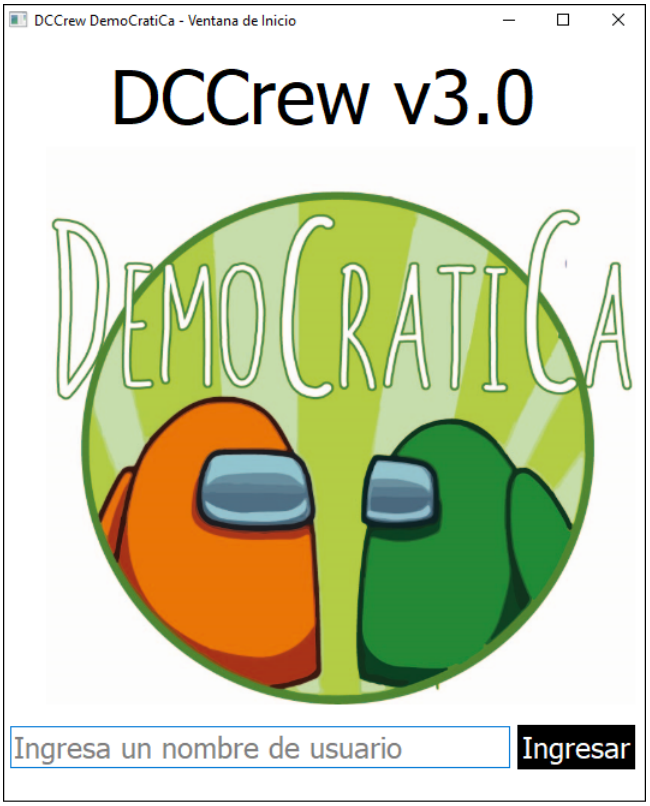

Figura 1: Ejemplo Ventana de inicio

Ventana de inicio encargada de inicializar al cliente con su nombre de usuario. Luego el cliente es redirigido a la **Sala de espera**.

2. **Sala de espera**

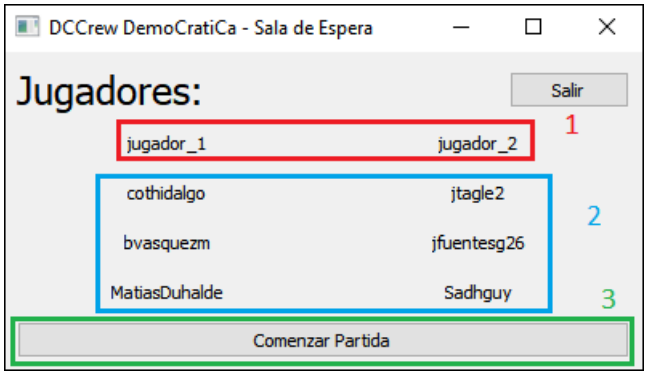

Figura 2: Ejemplo Sala de espera

Ventana correspondiente a la sala de espera del juego, donde (1) son los clientes 1 y 2 respectiva-

mente, (2) son los bots generados por el programa y (3) es el botón de inicio que puede ser apretado por **cualquiera de los dos clientes** para comenzar la partida. Luego se indicará el rol del cliente (impostor o tripulante) y se redireccionará a la **sala de votación**.

3. **Sala de votación**

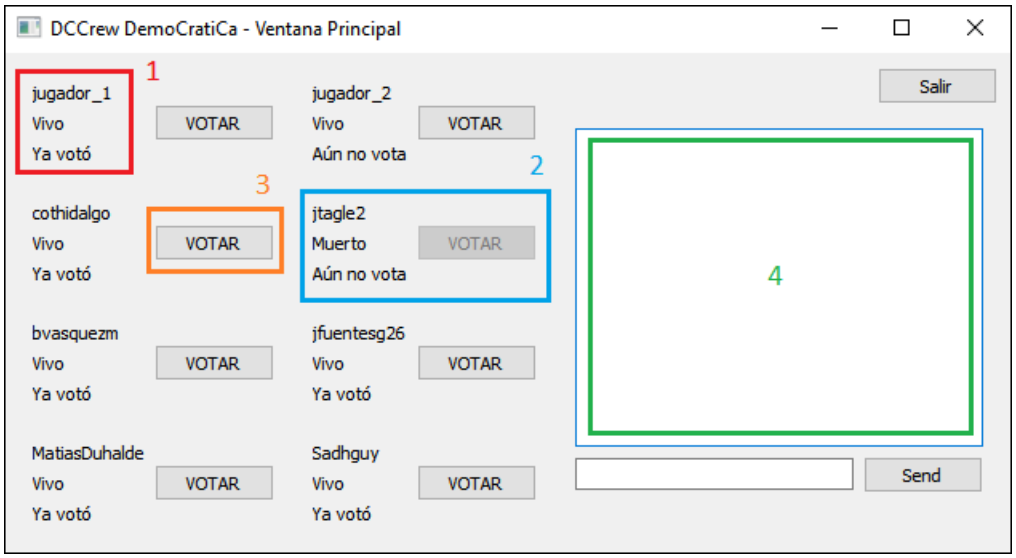

Figura 3: Ejemplo Ventana de votación

Ventana encargada del flujo general del juego, donde se llevan a cabo las votaciones y representan los estados de los jugadores. Donde (1) muestra el nombre, condición (vivo o muerto) y estado (si votó o no) del jugador, (2) representa un jugador eliminado del juego, (3) es el botón que acciona la votación sobre el jugador respectivo y (4) corresponde a la zona del chat, donde los mensajes de los **clientes** son desplegados. Seguido a la votación, se muestran los resultados de la misma y en caso de terminarse la partida, se indica el desenlace de los clientes (victoria o derrota).

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.
- Son libres de crear nuevos atributos y métodos que crean necesarios para el desarrollo de sus programas.

Objetivos

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