Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2020-1

18 de junio de 2020
Actividad Formativa

Actividad Formativa 06

Networking

Entrega

- **Lugar:** En su repositorio privado de GitHub, en la **carpeta** `Actividades/AF06/`
- **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**.

Nota: 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.

Introducción

Ahora que ya conoces a tus ayudantes y sus intereses, se vio que todos compartían una gran pasión: el arte. Se decidió que para terminar el semestre con todas las habilidades de un buen programador se explorarán los dotes artísticos de los ayudantes junto con los alumnos, fomentando la creatividad y el trabajo en equipo, en un programa hecho por los mismos estudiantes.

Para esto se creó ***DrawColorCanvas®***, programa que consiste en un gran lienzo en blanco, donde muchas personas pueden pintar píxeles de diferentes colores para crear una obra de arte en conjunto.

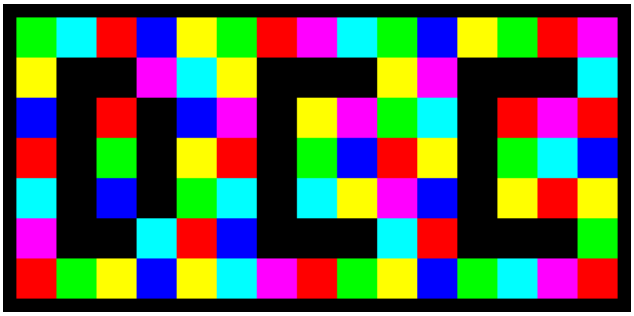

Figura 1: Logo DrawColorCanvas®hecho con DrawColorCanvas®.

Para implementar el programa se pidió tu ayuda como experto en *networking*, siendo tu misión lograr

conectar varios clientes con un mismo servidor, y así que todos juntos puedan dibujar de manera simultánea en un mismo lienzo, logrando una verdadera pieza artística moderna.

Flujo del programa

Networking, al igual que interfaces gráficas, se compone de dos principales secciones: el **Cliente** y el **Servidor**. La gran diferencia aquí, es que estos componentes suelen ser programas que funcionan paralelamente en diferentes máquinas y cada uno lleva sus procesos de forma independiente. La idea es que logres implementar estos programas de manera que se pueda establecer una conexión en base a un protocolo establecido, y pueda existir comunicación entre los procesos. Luego, te desafiamos a que extiendas la implementación, de manera que múltiples usuarios puedan interactuar a la vez, es decir, que hayan varios clientes conectados a un mismo servidor. Para esto podrás implementar primero la funcionalidad de solo un cliente conectado y luego ampliarlo a la conexión de múltiples clientes.

En el código de ***DrawColorCanvas®*** se te entrega una carpeta para el cliente y otra para el servidor, cada una con los archivos necesarios para su ejecución. Los roles y funcionalidades que tendrán en esta actividad se detallan a continuación:

- **Cliente:** El cliente es el programa que interactúa directamente con el usuario mediante una interfaz gráfica ya implementada. Este programa debe tomar la información que la interfaz entregue respecto a los cambios que el usuario ejecute en el tablero, y enviarla al servidor para su posterior procesamiento y distribución.
- **Servidor:** El servidor es un programa que funciona de forma paralela e independiente a los clientes y no interactúa con ningún usuario. Recibe la información de los clientes conectados, la almacena para generar un tablero común, y la distribuye a cada uno de los clientes para actualizar los tableros que ellos poseen localmente.

En esta interacción de cliente-servidor, el servidor está constantemente esperando nuevas conexiones, por lo tanto, cuando un cliente se conecta al servidor este está programado para avisar su llegada enviando un diccionario con el comando `"nuevo"` al momento de abrir la interfaz (`show()`). Cuando esto ocurre, el servidor responde enviando el estado actual del tablero (explicado en la sección Servidor), comenzando toda la dinámica de actualizaciones según los cambios de los usuarios conectados. De la misma forma, cuando el cliente termina la conexión (cerró la ventana `closeEvent()`), este envía al servidor un diccionario con el comando `"cerrar"` para indicar el término de la conexión.

Debido a esta configuración de interacciones, el siguiente orden de desarrollo te facilitará la implementación de ambas secciones.

CONSIDERACIÓN: Ten en cuenta que para esta actividad solo podrás comprobar el correcto funcionamiento del programa una vez que implementes Cliente en su totalidad y Servidor para al menos un cliente.

Cliente

En el cliente de *DrawColorCanvas®*tienes un interfaz incluida que te permitirá interactuar con el programa y con otros usuarios conectados al mismo servidor. Su estructura consta principalmente de un lienzo en blanco al centro (tablero) compuesto de 50×50 píxeles y una barra de color en la parte inferior con 8 alternativas para escoger. Para comenzar a dibujar, solo tienes que seleccionar tu color favorito y presionar en cualquier parte del tablero para dejar una marca de color sobre el píxel correspondiente.

Todo lo relacionado con la implementación del cliente se encuentra en la carpeta `cliente/`. Esto corresponde a los siguientes archivos:

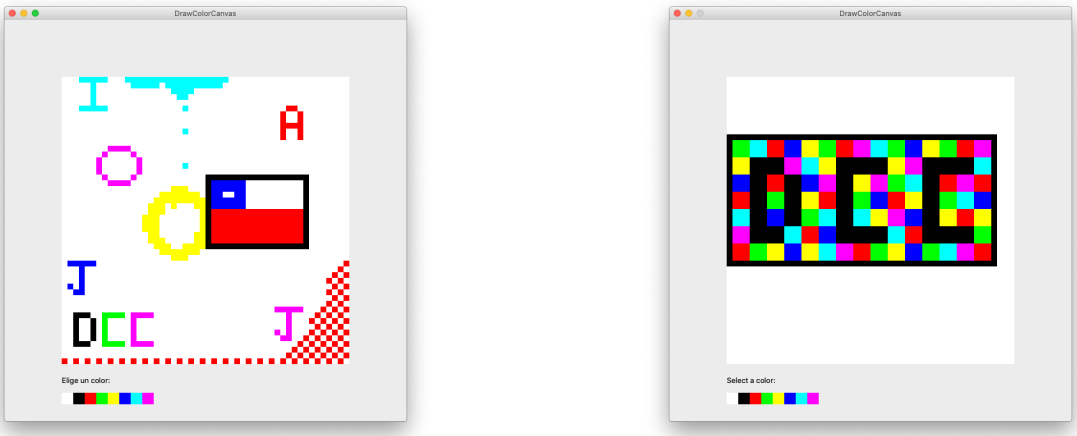

Figura 2: Ejemplo de interfaces de *DrawColorCanvas*®.

- `cliente.py`: Contiene la lógica del cliente. Esta se implementa principalmente dentro de la clase `Cliente`, la cual contiene atributos y métodos que permiten la comunicación con el servidor. Este será el único archivo sobre el cual deberás trabajar en esta parte.
- `draw_color_canvas.py`: Contiene toda la lógica de la interfaz gráfica del programa del cliente. Se conecta con el *back-end* por medio de señales. Puedes ejecutarlo para ver la interfaz, pero no será funcional. Este archivo **NO debes modificarlo**.
- `main.py`: Corresponde al archivo principal **a ejecutar**. En este, se inicializa la interfaz gráfica y se instancia la clase `Cliente`. Este archivo **NO debes modificarlo**.

Dentro del archivo `cliente.py` deberás modificar la clase `Cliente`, que contiene los siguientes métodos:

- `def __init__(self, host, port):` En este metodo se inicializa la clase Cliente junto con su interfaz, además se crea un *socket* y se conecta mediante el protocolo `TCP` al servidor. **Este método ya esta implementado y NO debes modificarlo**, y posee los siguientes atributos relevantes:
  - `self.socket_cliente`: Corresponde al *socket* que establece la conexión con el servidor.
  - `self.conectado`: Es un `bool` que indica si el *socket* aún está conectado al servidor. Es `True` si la conexión es estable y `False` si ya no hay conexión.
  - `self.senal_a_interfaz`: Es una señal que transporta información mediante un diccionario hacia la interfaz.
- `def escuchar_servidor(self):` Permite al cliente escuchar la información que el servidor envíe y transmitirla a la interfaz mediante una señal. Se encuentra (casi) vacío, **ya que debes completar este método**, dentro de la sección `try`, para que el cliente escuche **constantemente** al servidor. Mientras el cliente siga conectado (`self.conectado`) deberás:
  - Recibir el largo del mensaje, el cuál irá en los primeros 5 *bytes* del mensaje, serializado en *little endian* (`byteorder="little"`).
  - Pasar el largo del mensaje a int[1].
  - Luego, debes recibir el resto del mensaje en *chunks* de máximo 128 *bytes* cada uno.

[1]Recuerda que puedes deserializar un entero con el método `from_bytes` de `int`.

- Posteriormente, debes decodificar los *bytes* para obtener el *string* y deserializarlo para obtener el diccionario enviado por el servidor.
- Finalmente, debes enviar este diccionario a la interfaz a través de la señal `self.senal_a_interfaz`.

- `def enviar(self, mensaje):` Recibe un diccionario, que debe ser ser enviado al servidor. **Deberás completar este método** para que el servidor reciba el mensaje siguiendo el siguiente protocolo:
  - En primer lugar, debes serializar el mensaje como un *string* y codificarlo (covertirlo a *bytes*).
  - Debes obtener el largo del mensaje en 5 *bytes* y con `byteorder="little"`.
  - Finalmente debes enviar el largo del mensaje y el mensaje al servidor.
- `def enviar_a_servidor(self, dict_):` Es un método intermedio para separar las funciones de *back-end* y servidor, por lo que está conectado a la señal `senal_a_cliente` de la interfaz y maneja la información que esta entregue antes de enviarla al servidor con el método `self.send(dict_)`. **Este método ya esta implementado y NO debes modificarlo**.

Servidor

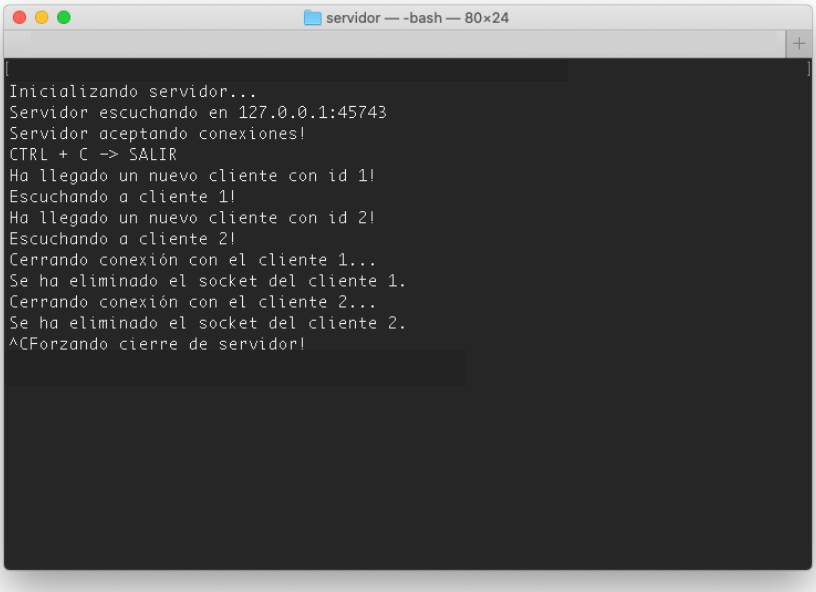

Figura 3: Ejemplo de servidor de *DrawColorCanvas*®.

En el servidor no habrá interacción con una interfaz, este funcionará en la consola de forma autónoma hasta que sea detenido con el comando `CTRL + C`.

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

- `servidor.py` Es el módulo que contiene la lógica principal del funcionamiento del servidor. En él encontrarás la clase `Servidor`, la cual contiene los atributos y métodos necesarios para la conexión con los clientes, que deberás completar para su correcto funcionamiento.
- `color_canvas.py` Este es un módulo intermedio que permite a la clase `Servidor` modelar el funcionamientos de las interfaces gráficas de los usuarios conectados y almacenar la información que ellos

envíen. Este archivo **NO debes modificarlo**.

- `main.py` Corresponde al archivo principal **a ejecutar**. En el, se instancia la clase `Servidor` y se inicia su funcionamiento para aceptar conexiones de clientes. Este archivo **NO debes modificarlo**.

Tu trabajo consiste en modificar la clase `Servidor` del módulo `servidor.py`, por lo que te dejamos una descripción de algunos de sus contenidos, que te servirán más adelante:

- `def __init__(self, host, port):` Este método se encarga de inicializar el servidor, creando su *socket* capaz de escuchar a usuarios e instancia un *DrawColorCanvas*®vacío, que será compartido entre los clientes que se conecten. Esta clase posee los siguientes atributos principales:
  - `self.socket_server`: Es el *socket* del servidor, desde el cual se debe aceptar conexiones.
  - `self.sockets_clientes`: Es un diccionario que servirá para almacenar los *sockets* de múltiples clientes.
  - `self.un_cliente`: Es un `bool` que indica si el servidor funcionará con un cliente (`True`) o con multiples (`False`).
  - `self.canvas`: Es una instancia de la clase `Canvas` de que permite modelar el tablero que los usuarios comparten. Esta instancia posee los siguientes métodos **ya definidos** que ocuparás durante la actividad:
    - `self.canvas.pintar_pixel(data)`: Es un método que permite pintar un píxel del tablero según lo que envíe el cliente. Recibe el argumento `data` que es un diccionario con toda la información del píxel a pintar.
    - `self.canvas.obtener_tablero()`: Es un método que permite obtener el estado actual del tablero, es decir, color de cada píxel, además no recibe argumentos y retorna un diccionario con toda la información lista para enviar a los clientes.

  Ya esta implementada y NO debes modificarla.
- `def enviar_respuesta(self, socket_cliente, respuesta):` Este método recibe el *socket* de un cliente, y un diccionario que contiene la respuesta a enviar. Si sólo se permite un cliente, se envía al cliente con `self.enviar` (explicado en la siguiente sección), mientras que si hay múltiples clientes, se envía a todos sus *sockets* con `self.enviar_a_todos` (explicado en la sección final). **Este método ya esta implementado y NO debes modificarlo**.

Conexión de un cliente

El servidor trabajará con **solo un cliente** cuando el atributo `self.un_cliente` sea `True`, valor que ya viene por defecto. En caso de que desees probar la conexión a múltiples clientes, deberás cambiar este atributo a `False`, pero para esta sección deberás dejarlo en `True`.

Para que el servidor soporte la conexión de un solo cliente, adicionalmente deberás completar los siguientes métodos como se indica:

- `def conectar_un_cliente(self):` Este método se encarga de aceptar la conexión de **un cliente**. En este método debes utilizar el *socket* del servidor para aceptar **un** cliente y su *socket* respectivo. Además debes informar en la consola que un cliente se ha conectado y comenzar a recibir información de inmediato, utilizando el método `self.escuchar_cliente`.
- `def escuchar_cliente(self, socket_cliente, id_cliente=0):` Este método permite que el servidor reciba información de un cliente. Recibe un *socket* y un *id* para identificar al cliente.

Este método debe ser capaz de recibir **constantemente** información desde el *socket* del cliente recibido, para lo que debes seguir los siguientes pasos:

1. Recibir el largo de la información total en los primeros 5 *bytes*. Los cuales son un `int` serializados con `byteorder="little"`[2].
2. Luego, en base al largo total de la información, debes recibir la respuesta en *chunks* de máximo 128 *bytes* cada uno.
3. Esta respuesta son los bytes de un *string* codificado, que representa a un diccionario serializado. Por lo tanto, debes decodificar los *bytes* en el *string* y luego des-serializarlo para obtener el diccionario que envió el cliente.
4. En este diccionario viene una llave `"comando"` la cual trae asociado un *string* que indica como deben continuar, este puede ser una de las siguientes opciones:
   - `"nuevo"` Este comando se envía apenas el cliente se conecta al servidor, y se abre la interfaz. Sirve para obtener el estado actual del canvas por primera vez. No se debe actualizar ningún pixel en el servidor, solamente se debe continuar al punto 5.
   - `"pintar"` Para este comando debes llamar al método ya implementado de la clase Canvas del Servidor `self.canvas.pintar_pixeles` y entregarle el diccionario que acabas de recibir.
   - `"cerrar"` Si este comando surge, entonces debes terminar de escuchar al cliente y pasar directamente al `finally` para que el programa termine la conexión. [3]
5. Por último debes recibir un diccionario desde el método `self.canvas.obtener_tablero()`, el cual será tu respuesta al usuario y debes enviarla a el(los) clientes respectivo mediante el método `self.enviar_respuesta(socket_cliente, respuesta)` que verifica cada caso.

- `def enviar(self, socket_cliente, mensaje):` Este método recibe un *socket* y un diccionario `mensaje`. Debes implementar los siguientes pasos para enviar el diccionario `mensaje` a través de la conexión con el `socket_cliente`:
  1. Primero debes convertir el diccionario a un *string*, el cual debes codificar en bytes.
  2. Luego debes calcular el largo de los *bytes* del mensaje ya codificado y enviar este largo (`int`) en 5 *bytes* con un `byteorder="little"`.[4]
  3. Por último, debes enviar todos *bytes* del mensaje ya codificado.

Propuesto: Conexión de múltiples clientes

Esta sección se deja propuesta ya que no entregará puntaje adicional. Sin embargo, te recomendamos hacerla para que entiendas cómo generalizar un servidor para un cliente de forma que funcione con múltiples clientes a la vez. Puedes hacerte una idea de la asignación de puntaje de la actividad revisando las secciones "Objetivos de la actividad" y "Objetivos adicionales".

Recuerda que el servidor trabajará con **múltiples clientes** solo cuando el atributo `self.un_cliente` sea `False`. Puedes revisar y modificar este valor en la línea 32 del método `__init__` de la clase `Servidor`. Recuerda reiniciar el servidor para que este cambio sea efectivo.

[2] Recuerda que puedes deserializar un entero con el método `from_bytes` de `int`.
[3] Recuerda que `return` termina la ejecución de una función, por lo tanto, no podrías llegar al `finally`.
[4] Puedes serializar un `int` con su método `.to_bytes`.

- `def conectar_varios_clientes(self):` En este método debes aceptar conexiones de clientes **constantemente**. Para ello, cada vez que un cliente establezca una conexión con el servidor tienes que aceptar su *socket* y asignarle un *id* único. Esta información debes guardarla en el diccionario `self.sockets_clientes` del Servidor, donde la llave es el *id* asignado y el valor es el *socket* del cliente.

  Una vez guardado el *socket*, debes comenzar a recibir información desde este cliente mediante el método `self.escuchar_cliente` que ya implementaste, el cual recibe el *socket* y el *id* como argumentos. Recuerda que ahora debes escuchar a múltiples clientes simultáneamente, por lo tanto, debes buscar la forma de que esto no impida conectar y escuchar a varios clientes de forma paralela, pero su ejecución no debe ir más allá del funcionamiento del programa.

  Una vez que un cliente se ha conectado y ya está recibiendo información, debes esperar a que otro cliente se conecte y repetir los pasos. **No hay máximo de clientes conectados**.

- `def enviar_a_todos(self, mensaje):` Este método envía el mensaje a todos los *sockets* actualmente conectados. **Este método ya esta implementado y NO debes modificarlo**.

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.
- Para cerrar el servidor debes presionar `CTRL + C` en cualquier momento durante su ejecución. Deberías ver un mensaje informando el cierre del servidor siempre que no haya ocurrido un error.
- 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 de la actividad

Como podrás notar, reemplazamos la sección "Requerimientos" por esta sección, con la intención de señalar lo que buscamos que aprendas al realizar esta actividad. También queremos entregarte un acercamiento más directo a los criterios que se utilizarán para la asignación de puntaje de la actividad.

- Implementar un **Cliente** capaz de recibir y enviar mensajes dado un protocolo específico.
- Implementar un **Servidor** capaz de recibir, manejar y enviar mensajes a un cliente dado un protocolo específico.

Objetivos adicionales

- Implementar un **Servidor** capaz de recibir, manejar y enviar mensajes a varios clientes a la vez.
- Establecer una conexión coherente entre un **Servidor** y múltiples **Clientes**.