Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

**IIC2233 Programación Avanzada (2019-1)**

# Tarea 03

## Entrega

- **Avance de tarea**
  - **Fecha y hora:** miércoles 12 de junio de 2019, 20:00
  - **Lugar:** GitHub — Carpeta: `Tareas/T03/`
- **Tarea**
  - **Fecha y hora:** domingo 23 de junio de 2019, 20:00
  - **Lugar:** GitHub — Carpeta: `Tareas/T03/`
- `README.md`
  - **Fecha y hora:** lunes 24 de junio de 2019, 20:00
  - **Lugar:** GitHub — Carpeta: `Tareas/T03/`

## Objetivos

- Aplicar conocimientos de creación de redes para comunicación efectiva entre computadores.
- Diseñar e implementar una arquitectura cliente-servidor.
- Aplicar conocimientos de manejo de *bytes* para crear o modificar un tipo de archivo preestablecido.

# Índice

# 1. Introducción a Timbiriche 99

Luego del éxito del juego **Tetris 99**, el DCC decidió lanzar su propio juego del estilo *battle royale*: **Timbiriche 99**, también conocido como "El juego de líneas y cuadrados sin nombre". Para esto te piden a ti, como experto en *networking*, que implementes este juego. Éste debe permitir a usuarios: encontrar partidas de Timbiriche 99 abiertas, comunicarse mediante mensajes con otros usuarios, y jugar una partida simultáneamente con múltiples usuarios.

# 2. Timbiriche 99

El juego se lleva a cabo sobre un tablero, el cual contiene $N \times M$ puntos. Por turnos, los jugadores colocan líneas horizontales o verticales que unen dos puntos no conectados anteriormente y que sean ortogonalmente adyacentes, es decir, los puntos deben estar juntos y la línea que los una no puede estar en diagonal ni pasar por sobre otros puntos.

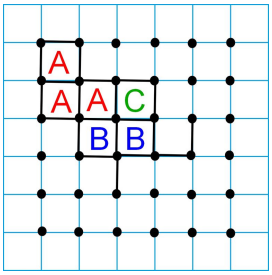



Figura 1: Ejemplo de tablero de $6 \times 6$ del juego.

Si un jugador, luego de colocar una línea, logra formar un cuadrado (véase Subsección 2.1), entonces gana un punto y le toca poner otra línea. El turno del jugador continuará hasta que coloque una línea que no complete un cuadrado. Entonces empieza el turno del siguiente jugador. El juego termina cuando no se pueden colocar más líneas y el ganador es aquel jugador que tenga el mayor puntaje, es decir, aquel que haya formado más cuadrados.

Deberás crear un programa que permita jugar a varios usuarios conectados en la misma red local utilizando *networking*. Deberás modelar el juego siguiendo el modelo cliente-servidor, en donde cada usuario actúa como cliente y para jugar, debe conectarse y comunicarse con el servidor del juego.

## 2.1. Formar un cuadrado

Se considera que un jugador forma un cuadrado cuando agrega una línea que "encierra" un área de tamaño $1 \times 1$, la cual tiene líneas en todos sus lados.

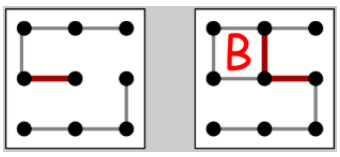



Figura 2: El jugador B agregó una línea que encierra un cuadrado, por lo que recibe un punto y juega nuevamente.

## 3. Flujo de juego

Cuando un jugador inicia el programa, deberá abrirse una ventana que le permita acceder con su nombre de usuario. El juego debe verificar la existencia del nombre de usuario y corroborar que ningún otro jugador con el mismo nombre se encuentre conectado en ese momento. Una vez verificado, se abre una ventana con la foto de perfil del jugador y una lista de salas de espera disponibles, además de una opción para crear una sala nueva y otra para cambiar de usuario. La foto de perfil se puede cambiar y se puede seleccionar la opción de verse con **filtro dibujo**, el cual será explicado más adelante.

En la ventana anterior deben verse todas las salas disponibles, junto con su nombre y la cantidad de usuarios conectados en cada sala. Las salas de espera son el lugar donde los jugadores se reúnen antes de iniciar una partida. Éstas poseen un máximo de 15 jugadores cada una. Haciendo *click* en el botón correspondiente de alguna sala, se puede acceder a ella.

Una vez dentro de la sala, el usuario puede ver a los demás jugadores que se encuentran esperando y puede interactuar con ellos mediante un *chat*. En cada sala existe un jefe, quien inicialmente es el creador de la sala. Todos los jugadores, menos el jefe de la sala de espera, verán un aviso que diga "esperando juego". El jefe de la sala tendrá un botón que dice "Iniciar partida", el cual (como dice su nombre) le permite iniciar una partida del juego para todos los usuarios que se encuentren en su sala. Si el jefe sale de la sala antes de iniciar el juego, se nombra a otro jugador cualquiera como el jefe de sala, para que éste pueda iniciar el juego.

Al iniciar el juego deberá mantenerse el *chat* que los usuarios tenían en la sala de espera, para que puedan comunicarse (amablemente) durante la partida. Junto con el *chat* se encuentra la ventana de juego, donde se desarrolla la partida, según lo indicado por las reglas de Timbiriche 99. Además, cuando un jugador decide crear una sala nueva, el servidor le asigna un nombre a esta sala, asegurándose de que éste sea único y descriptivo[1].

Una vez terminada la partida, el jefe puede elegir entre jugar otra ronda o terminar. Si es que selecciona terminar, la sala se cierra y todos los usuarios que se encontraban dentro de ella son llevados a la ventana de salas, para poder seleccionar otra y así poder seguir jugando.

## 4. *Networking*

¡Así es! Como has de suponer, el Timbiriche no se juega solo, sino que funciona en base a una interacción entre una cantidad arbitraria de jugadores. En esta tarea deberás proporcionar dicha funcionalidad implementando una arquitectura **cliente-servidor**. Para esto, tienes que hacer uso del *stack* de protocolos de red TCP/IP a través de la biblioteca `socket`. Esta biblioteca te proporcionará las herramientas necesarias para administrar la conexión entre dos computadores conectados a una misma red local[2].

El servidor es el primero que debe ejecutarse. Luego de empezar, queda a la espera de que distintos clientes se conecten a él para comenzar partidas. Es importante destacar que en el modelo cliente-servidor, la comunicación ocurre **exclusivamente** entre los clientes y el servidor[3], es decir, nunca directamente entre dos clientes.

[1] Por ejemplo: "Sala 03". No un conjunto de letras al azar.
[2] Una red local es la interconexión de varias computadoras, puedes ver más información en Red de área local.
[3] Nótese que dice **el** servidor (en singular), pues debe existir exactamente uno (ni más ni menos).

### 4.1. La conexión

El servidor abrirá un *socket* haciendo uso de los datos encontrados en el archivo `server/parametros.json`. Tal como sugiere su extensión, éste será de tipo `JSON` y seguirá el formato mostrado a continuación. Por otra parte, el cliente deberá conectarse al *socket* abierto por el servidor haciendo uso de los datos encontrados en `client/parametros.json`. Tienes la libertad de hacer que éstos los obtenga del archivo, o bien a través de la interfaz gráfica (explicado en Interfaz gráfica).

```
{
    "host": <dirección_ip>,
    "port": <puerto>,
    ...
}
```

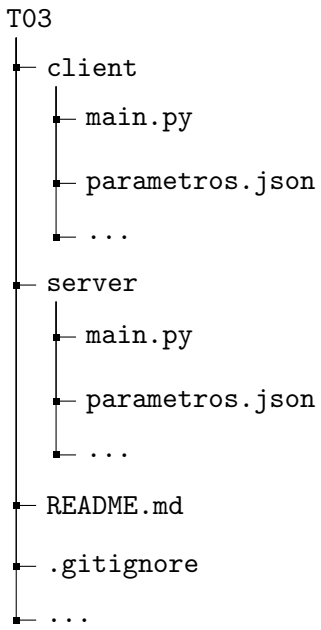
Cabe destacar que, luego de formalizar dicha conexión, la manera en la que implementes el envío de mensajes entre el cliente y el servidor queda totalmente a tu criterio.

### 4.2. Arquitectura cliente-servidor

En esta sección se presentarán algunas consideraciones que deberás aplicar en la elaboración de tu tarea. Básicamente, funciona así: si está acá es porque debes seguirlo al pie de la letra; si no se encuentra especificado puedes suponer que queda a tu criterio.

#### 4.2.1. Separación funcional

El cliente y el servidor deben estar separados. Esto significa que tienen que estar contenidos en directorios (carpetas) distintos: uno llamado `client` y el otro llamado `server`. Cada uno de ellos deberá contener un archivo llamado `main.py` que ejecute el programa indicado por el nombre del directorio que lo contiene, en conjunto con todos los módulos y archivos que hayas creado para su correcto funcionamiento. A continuación se muestra un diagrama con la estructura descrita.



Sin embargo, como la separación es funcional, a pesar de que el cliente y el servidor comparten un mismo directorio padre (`T03`), estos tendrán que funcionar de manera **completamente independiente** (tal como si estuvieran en computadores distintos). Es decir, que ningún archivo del cliente podrá depender de ninguna manera de alguno de los archivos del servidor y viceversa.

En la siguiente figura se muestra cómo debes realizar la separación entre los distintos componentes de tu programa:

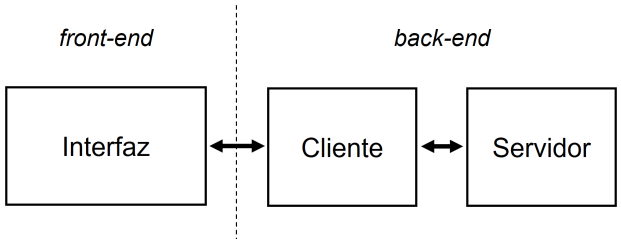



Figura 3: Separación cliente-servidor y *front-end-back-end*.

La comunicación entre el *back-end* y el *front-end* se debe realizar mediante señales, mientras que la comunicación entre el cliente y el servidor debe realizarse mediante *sockets*.

### 4.2.2. Roles

A continuación se especifican cuáles funcionalidades de tu tarea deben ser manejadas por el servidor, y cuáles deben ser manejadas por el cliente. Se evaluará que tanto el servidor como el cliente manejen únicamente las funcionalidades que le corresponden.

**Roles del servidor.**

1. El servidor es el motor de tu aplicación. Todo lo que tiene que ver con la lógica del juego y el procesamiento de datos deberá ser delegado a él. Por ejemplo, es el encargado de verificar que las jugadas recibidas sean válidas, o que el nombre de usuario de un jugador que intenta conectarse no esté siendo utilizado por alguien más.
2. El servidor debe ser el que recibe un mensaje o una acción ejecutada por cualquier jugador y distribuirlo a sus compañeros, de forma que todos los jugadores reciban la misma información, y sus interfaces sean consistentes entre sí.
3. Como a (casi) nadie le gustaría crearse una cuenta y que al día siguiente ésta deje de existir, es labor de tu servidor que esto **nunca** suceda. El servidor deberá manejar una base de datos donde se pueda crear, almacenar y actualizar la información necesaria para que los datos de los usuarios registrados no se pierdan.
4. El servidor **no tendrá** una interfaz gráfica asociada.

**Roles del cliente.**

1. El cliente deberá encargarse de mostrar la interfaz gráfica, para que el usuario pueda interactuar y jugar con ella.
2. El cliente deberá encargarse de enviar las acciones del usuario al servidor.
3. El cliente deberá encargarse de recibir y procesar los mensajes del servidor.

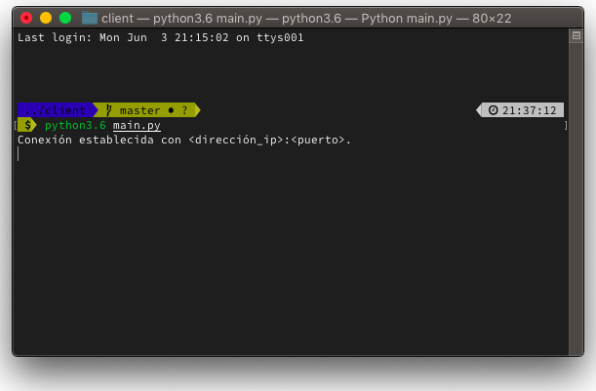
# 5. Interfaz gráfica

Para una experiencia de juego más placentera, deberás implementar una interfaz gráfica que guíe al usuario durante todo el transcurso del juego, es decir, desde que decide ingresar a la aplicación hasta que sale de ella.

Cabe destacar que todo lo descrito en esta sección hace referencia a una interfaz gráfica que deberán implementar sólo para el **cliente**, haciendo uso de la librería `PyQt5`. Sin embargo, las figuras que se muestran a continuación son únicamente de referencia; su implementación no tiene por qué parecerse a ellas.

## 5.1. Conexión con el servidor.

La primero que necesitamos para poder comenzar a jugar es conectarnos al servidor a través del cual interactuaremos con los demás jugadores. Para esto, debemos conocer tanto su dirección IP como el puerto por el que recibirá las conexiones. El cómo hacer esto queda a tu criterio, por lo que podrás implementarlo leyendo un `JSON` (explicado en *Networking*) o (si quieres ser más *hacker*) recibiendo dichos parámetros a través de la interfaz (véase la Figura 4). La manera en la que decidas hacerlo no tendrá ningún impacto en tu nota, siempre y cuando permita conectar a ambos agentes correctamente.




(a) A través de la terminal. Los datos se obtienen del archivo `parametros.json`.


(b) A través de la interfaz gráfica. Los datos los ingresa el usuario.

Figura 4: Dos maneras igualmente válidas de conectarse al servidor.

### 5.2. Autenticación del usuario

Luego de establecer una conexión exitosa, se deberá ingresar un nombre de usuario para comenzar a jugar. Al intentar realizar esta acción, el usuario se encontrará en alguna de las siguientes situaciones.

1. El nombre de usuario ingresado **no se encuentra registrado**. ¡Felicidades! Esto significa que es la primera vez que un cliente ingresa al juego con ese nombre de usuario, por lo tanto, el servidor deberá instanciarlo y almacenarlo en su base de datos.
2. El nombre de usuario ingresado **sí se encuentra registrado** y actualmente **está desconectado**. Esto implica que algún cliente ya ha ingresado al juego haciendo uso de ese nombre de usuario. No obstante, no hay ningún cliente actualmente en línea utilizándolo, por lo que se deberá proceder con normalidad.
3. El nombre de usuario ingresado **sí se encuentra registrado** y actualmente **está conectado**. Esto significa que el nombre de usuario ya fue registrado anteriormente por algún cliente. Sin embargo, este se diferencia del caso anterior, pues en este preciso momento ya hay otro cliente en línea haciendo uso del mismo nombre de usuario. Como no pueden haber dos jugadores conectados que se llamen igual, el servidor tendrá que informar al cliente de esta situación, y no permitir su ingreso, hasta que envié un nombre de usuario que no este conectado. Además, se deberá notificar de esta situación a través de un mensaje de error en la interfaz.

Como habrás notado en el punto número 3, esta manera de conectarse no es muy segura, ya que faltan las **contraseñas**. En caso de que desees implementarlas —por un par de décimas más— revisa este *bonus*.

### 5.3. Salas

Timbiriche 99 debe ser capaz de manejar múltiples partidas a la vez, para lo que deberás implementar **salas de juego**. Una vez que el usuario logre iniciar sesión, se deberá contar con una ventana nueva que permita visualizar todas las **salas disponibles** de manera clara[4]. Además, para cada sala se debe mostrar la **cantidad de jugadores** que han ingresado a ella y el **nombre de esta**. Toda esta información debe ser fácilmente accesible y tiene que ser actualizada en tiempo real. Por ejemplo, si una sala es creada por un usuario, esta debe aparecer en todos los usuarios que se encuentran conectados al servidor. Si una sala es cerrada, entonces debe desaparecer de la lista de salas de todos los usuarios.

Naturalmente, cada sala deberá contar con la opción de **unirse** a ella. Esta opción deberá bloquearse automáticamente en caso de que la sala esté llena, es decir, que el número de jugadores que la componen haya llegado a su máximo, el cual es de **15 jugadores** por sala.

Por último, se deberá contar con una sección llamada "Perfil del usuario", en la que se mostrará el **nombre de usuario** y **foto de perfil** en cuestión. En caso de no existir foto de perfil, debe aparecer un selector de archivos para subir una imagen[5] y que se envié al servidor. Además, deberá estar presente un **botón para aplicar el filtro dibujo** a la foto de perfil, explicado en Filtro Dibujo.

#### 5.3.1. Creación de sala

Además de todas las salas disponibles, deberá existir una opción para crear una nueva sala. Una vez creada, debe ser visible en la lista de salas disponibles. La persona que cree la nueva sala será asignada

[4] Se recomienda hacer uso de `QScrollArea` o algún otro elemento de la librería `PyQt` que sea *scroleable* para listar las salas.
[5] Se recomienda hacer uso de `QFileDialog` de la librería `PyQt` para el selector de archivos.

como jefe de esta, hasta que se desconecte o salga de la sala. El nombre de la sala creada es decidido por el servidor, que debe asegurarse de que no exista otra sala creada con ese mismo nombre.

### 5.3.2. Preparación de la partida

Las salas siempre deberán tener un único jefe. Este es asignado de manera automática, siendo este el que haya creado la sala. En caso de que el jefe actual se salga de la sala o del juego, este rol deberá asignarse a otro usuario. La forma de decidir quién será el próximo jefe queda a tu criterio. Además, la sala se deberá cerrar si esta llegase a quedar sin jugadores, en esta situación la sala deja de estar visible.

Es importante que en tu interfaz sea claramente visible en todo momento el **nombre de usuario del jefe** de la sala. Además, deberá existir un **botón para iniciar la partida** (que sólo podrá ser presionado por el jefe de la misma). Al momento de presionar dicho botón tendrá que aparecer una cuenta regresiva de cinco segundos, al final de la cual comenzará el juego. Para el desarrollo del juego, puedes hacer que la misma ventana se transforme o puedes crear una nueva.

### 5.3.3. Partida

Una vez comenzada la partida, se deberá mostrar en todo momento los **nombres de usuario** y **puntaje** de cada uno de los jugadores. Dónde y cómo mostrar esta información queda totalmente a tu criterio. El único requisito es que los elementos no se superpongan con el juego en sí, de manera tal que este se pueda llevar a cabo de manera fluida y sin interrupciones. Y lo más importante: en esta misma ventana se deberá proveer la interacción con el **tablero**. Esta ventana deberá mostrar de manera clara el estado actual del juego. Esto incluye los vértices y aristas del tablero, además de los cuadrados formados por los jugadores y sus identificadores correspondientes.

¿Qué es un identificador? Una vez que un jugador logre encerrar un cuadrado, éste debe quedar marcado con un **identificador** (único para cada jugador), que debe ser visible para todos los usuarios. La naturaleza de este identificador queda a tu criterio, siendo lo importante que te asegures de que sea único para cada usuario dentro de la sala y que sea claramente identificable[6].

Los parámetros que determinan el número de puntos en el juego, $N$ y $M$, deben estar definidos en el archivo `parametros.json` dentro de la carpeta del servidor. El servidor indica a los clientes el número de puntos de la partida.

La partida se subdivide en turnos cíclicos. Es decir, que cada vez le toca a un jugador distinto, hasta que todos hayan jugado la misma cantidad de veces. Luego, se repite el ciclo. Es importante destacar que cada uno de estos turnos tiene una duración máxima en tiempo (que debe ir definida en `server/parametros.json`), y en la interfaz deberá verse una **cuenta regresiva** que indique el tiempo que va quedando del turno en curso en tiempo real. En caso de que dicho tiempo se agote y el usuario no haya realizado su jugada, este la perderá y comenzará de manera inmediata el turno del siguiente jugador.

## 5.4. *Chat*

Al unirse a una sala, se deberá mostrar un *chat*[7] con los participantes de esa sala. Esto significa que cada sala debe tener un *chat* propio, donde sólo podrán participar quienes estén en la sala.

[6] Puedes usar colores, números, fotografías, iconos, iniciales, letras (como en la Figura 1), *emojis* o lo que tú quieras.

[7] En esta ayudantía se implementa un *chat*, pero sin utilizar señales. ¡Recuerda citar correctamente si es necesario!

Los mensajes deberán ser mostrados en el formato `NombreDeUsuario: Mensaje`. Dentro de este *chat* es donde deberás implementar el comando `\chao NombreDeUsuario`, el cual comenzará una votación para la expulsión del jugador `NombreDeUsuario` de la sala. La expulsión debe hacerse efectiva en caso de que más de la mitad de los jugadores de la sala escriban el mismo comando en contra de ese jugador.

Cada acción de los jugadores debe quedar registrada por un mensaje en el *chat* de la sala, como también los saltos de turno de quienes no hicieron alguna jugada. Este mensaje debe ser enviado por el servidor, y en lugar de estar identificado por el nombre de algún usuario, se debe identificar con la etiqueta de servidor.

A modo de ejemplo, el chat de una sala podría verse de la siguiente manera:

```
iqacevedo: Esperen a ver mi jugada
hernan4444: Probablemente sea mala
servidor: iqacevedo ha realizado una jugada
hernan4444: No se si alcanzo a jugar, estoy ocupado
servidor: hernan4444 ha perdido su turno, debido a que se le acabo el tiempo
servidor: entamburini acaba de encerrar un cuadrado y gana un punto
```

Figura 5: Chat de sala de juego

## 6. Filtro Dibujo

Tu programa debe permitir que el usuario pueda aplicar filtros a las imágenes. El filtro que debes aplicar es del tipo **dibujo**, y como *bonus* tendrás la opción de crear un filtro que permita la visualización de imágenes para daltónicos. El filtro dibujo consiste en hacer una modificación de la imagen original, para que ésta se pueda ver como un *boceto* (ver Figura 6).

(a) Imagen original.

(b) Imagen con filtro.

Figura 6: Aplicación del filtro dibujo.

La aplicación de este filtro requiere que realices varias operaciones matriciales sucesivas a la matriz de píxeles RGB, para que luego sea guardada y pueda ser abierta por tu ordenador. Para lograr esto, debes seguir las instrucciones en la sección **Filtro Dibujo** del documento `T03 - PNG y Filtros.pdf`

En caso de que no implementes la foto de perfil, puedes crear una función llamada `filtro_dibujo` que recibe dos argumentos: (1) `path_original` es la ruta de la imagen que se le desea aplicar el filtro; y (2) `path_dibujo` es la ruta donde la función debe guardar **una nueva imagen** con el filtro aplicado. Si este es el caso debes indicarlo en tu `README.md`, junto con detallar donde se encuentra la función.

## 7. Manejo de *bytes*: formato PNG

En esta tarea tendrás que incorporar manejo de *bytes*, aplicado a los archivos de extensión `.png`. Para facilitar esta parte, se te entregará una librería llamada `pixel_collector.py`. De este archivo, deberás usar la función `get_pixels(image_path)`, donde `image_path` es la ruta un archivo de imagen, para leer una imagen y transformarla a matriz RGB (se explicará más adelante). Por lo tanto, deberás implementar manejo de *bytes* para transformar matrices RGB a archivos PNG. Para poder utilizar la librería `pixel_collector.py` debes instalar la librería `pillow`. Puedes instalarla escribiendo en la consola:

```
$ pip3 install Pillow
```

Para facilitar la escritura de este tipo de archivos, las instrucciones de manejo de *bytes* se encuentran en el documento `T03 - PNG y Filtros.pdf`.

Es importante destacar que el servidor deberá guardar las fotos de perfil de los usuarios en formato PNG. Por lo que al recibir una nueva foto, debe obtener la matriz RGB, y posteriormente guardarla en formato PNG.

En caso de que no implementes la foto de perfil, puedes crear la función `guardar_png(path_original, path_destino)`, donde el argumento `path_original` es la ruta de la imagen original, y el argumento `path_destino` es la ruta donde la función debe guardar **una nueva imagen** en formato PNG. Si este es el caso debes indicarlo en tu `README.md`, junto con detallar donde se encuentra la función.

## 8. `.gitignore`

Para esta tarea **el archivo** `pixel_collector.py` **deberá ser ignorado** con un `.gitignore` que deberá estar dentro de tu carpeta `T03`. Puedes encontrar un ejemplo de `.gitignore` en el siguiente *link*. Tampoco deberás subir el enunciado.

Se espera que no se suban archivos autogenerados por programas como PyCharm, o los generados por entornos virtuales de Python, como por ejemplo: la carpeta `__pycache__`.

Para este punto es importante que hagan un correcto uso del archivo `.gitignore`, es decir, los archivos no **deben** subirse al repositorio debido al archivo `.gitignore` y no debido a otros medios.

## 9. Avance de tarea

El avance de esta tarea corresponde a implementar el *chat* del juego. En particular, se espera que cumpla con:

- Tener implementada la arquitectura servidor/cliente.
- Conectarse con un número arbitrario de clientes.
- Utilizar señales.
- Tener implementado el comando **\chao**.

Notar que no es necesario crear el resto de la interfaz del juego para realizar este avance, solo se requiere un *chat* funcional.

# 10. *Bonus*

En esta tarea podrás realizar todos los *bonus* que quieras, pero a lo más podrás obtener **5 décimas** de bonificación. Solo se otorgará puntaje si un *bonus* es **realizado en su totalidad**. Si es que un *bonus* no está completo, ya sea por un pequeño error, o porque faltó implementar algo, no se dará puntaje. Finalmente, debes indicar en tu `README.md` cuáles *bonus* implementaste, en caso contrario no se considerarán realizados.

Para poder optar a los bonus, **debes obtener una nota igual o mayor a 4.0** en la tarea, antes de considerar los descuentos.

## 10.1. *Chat* con Emojis (2 Décimas)

Para obtener este *bonus* deberás agregar **Emojis** al *chat* del juego. Para enviar emojis los usuarios deben escribir el nombre del emoji entre dos puntos, por ejemplo:


Figura 7: *Bonus* Emojis.

Para este *bonus* puedes agregar todos los emojis que estimes necesarios y con los nombres e imágenes que quieras. Los requisitos son los siguientes:

1. Al enviar un emoji este se debe ver bien tanto como para el que lo envía como para todos los que lo reciben.
2. Se debe agregar un archivo `emojis.py` que contenga un diccionario de la forma:

   ```
   { "nombre_emoji": unicode_emoji }
   ```

   Este diccionario debe controlar todos los emojis que se pueden mostrar, es decir, si se agrega un nuevo emoji a este diccionario el programa tiene que detectarlo automáticamente.

Se puede revisar este *link* para obtener el *unicode* de emojis.

## 10.2. Poder Especial (3 Décimas)

Para obtener este *bonus* deberás implementar un **Poder Especial** al juego. Este consiste en dar la opción de **saltar el turno del jugador que actualmente está jugando**. Debe ser implementado como un botón que solo se puede apretar cuando es el turno de otro jugador. Si cualquiera de los jugadores que no están jugando aprieta el botón, el turno del jugador actual termina inmediatamente. Cada jugador solo puede utilizar este poder **tres veces** por juego.

Finalmente el servidor debe indicarle a todos los jugadores mediante el *chat* que un poder ha sido utilizado de la forma: “jugador1 saltó el turno de jugador2”.

### 10.3. *Undo* (3 Décimas)

Para obtener este *bonus* deberás implementar un **Botón *Undo***, es decir, un botón que permita deshacer la última jugada hecha en tu turno. Una vez hecha tu jugada, puedes utilizar este botón siempre y cuando no haya pasado ninguna de las siguientes cosas:

1. Hayan trascurrido dos segundos o más.
2. El siguiente jugador haya hecho su jugada.

Cuando un jugador utiliza su *undo*, se reinicia el tiempo límite de su turno.

Finalmente, solo puedes realizar *undo* una vez por turno y cuando lo hagas el servidor debe indicarle a los jugadores esto mediante el *chat* lo sucedido, de la forma: "jugador1 ha revertido su jugada".

### 10.4. Contraseñas (2 Décimas)

Para obtener este *bonus*, deberás implementar un sistema de contraseñas con codificación al juego. Esto significa que antes de poder ingresar un usuario debe tener dos opciones distintas con su propia interfaz, iniciar sesión y registrarse. En el caso de que un usuario quiera registrarse debe ingresar un nombre que no exista, es decir, que no haya sido registrado por el servidor anteriormente, y una contraseña con al menos 6 caracteres (no se debe permitir el registro si no se cumplen están condiciones). En el caso que el usuario quiera iniciar sesión, debe ingresar su nombre y su contraseña.

Finalmente, deberás guardar la contraseña en el servidor, pero de forma **codificada**[8]. Así, cada vez que un usuario quiera ingresar, simplemente debes codificar la contraseña que entregó, enviar la versión codificada de la contraseña y que el servidor la compare con la versión guardada en él. En caso de que ingrese una contraseña errónea, se le debe informar su error.

### 10.5. Servidor y Cliente Robusto (3 Décimas)

Para obtener este *bonus*, deberás lograr que tanto tu servidor como tu cliente sean robustos. Esto quiere decir, que si uno de los dos programas deja de funcionar (ya sea debido a pérdida de internet, corte eléctrico, etc.) el otro maneje correctamente la situación.

En caso de que el servidor deje de funcionar, todos los clientes deben mostrar un mensaje informando la pérdida de conexión con el servidor, dando la opción para cerrar el programa.

En caso de que un cliente deje de funcionar, el servidor debe detectar lo sucedido y seguir funcionando normalmente. También debe mostrar un mensaje en consola indicando "El usuario X se desconecto", donde X es el nombre de usuario.

Para este *bonus* pueden probar realizando *keyboard interrupt* en la consola del cliente o del servidor. También pueden probar cerrando la interfaz de un cliente.

### 10.6. Filtro Daltonismo (4 Décimas)

Para obtener este *bonus*, deberás implementar un **filtro de daltonismo**, es decir, un filtro que cambie una imagen normal a una imagen que una persona daltónica pueda ver correctamente. Sin embargo hay tres tipos de daltonismo, ejemplificados a continuación:

[8] Para esto les recomendamos utilizar la librería `bcrypt`.

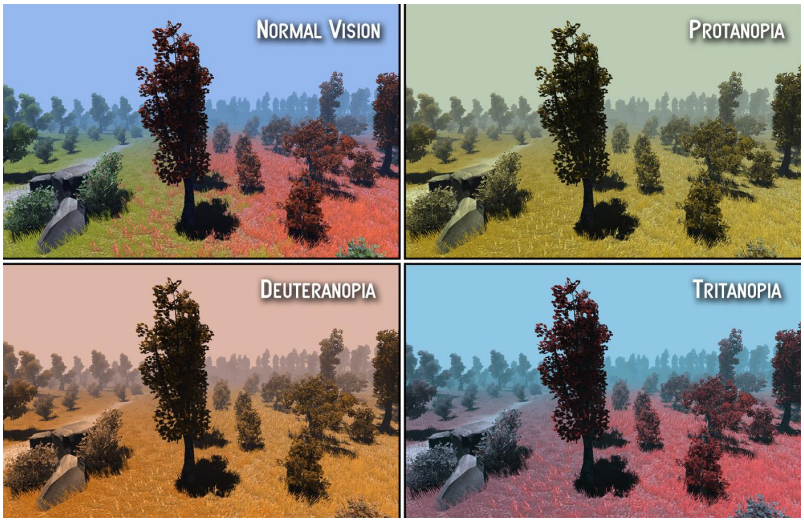



Figura 8: Diferentes tipos de daltonismo.

En este caso deberán hacer un filtro de **Deuteranopia**. Específicamente en la sección de perfil dentro de la vista de salas debe haber un segundo botón que permita aplicar este filtro.

### 10.7. Timbiriche 99 Diagonal (5 Décimas)

Para obtener este *bonus* deberás implementar un cambio fundamental, ahora deberá soportar **líneas diagonales**. Las diagonales pueden ser en cualquier dirección, o solo en una, según lo decidas. Por ejemplo:

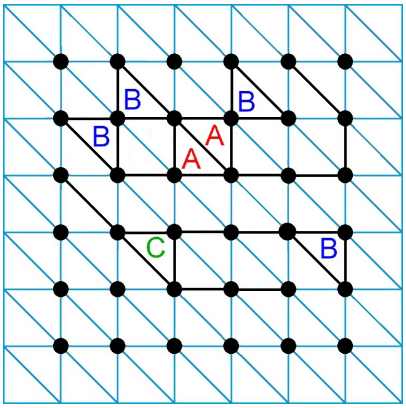



Figura 9: Tablero de $6 \times 6$ con líneas diagonales.

Cabe recalcar que si se encierra un cuadrado **no se considerará** puntaje, **se debe encerrar un triangulo**. Finalmente, el puntaje se calcula según la cantidad de triángulos.

## 11. Importante: corrección de la tarea

Para esta tarea, el carácter funcional del juego será el pilar de la corrección, es decir, **solo se corrigen tareas que se puedan ejecutar**. Por lo tanto, se recomienda hacer periódicamente pruebas de ejecución de su tarea y `push` en sus repositorios de versiones funcionales de su programa.

Cuando se publique la distribución de puntajes, se señalará con un color rojizo cada ítem que será evaluado a nivel de código, todo aquel que no esté pintado en rojizo significa que será evaluado si y solo si se puede probar con la ejecución de su tarea. En tu `README.md` deberás señalar el archivo y la línea donde se encuentran definidas las funciones o clases relacionados a esos ítems.

Finalmente, se recomienda el uso de *prints* para ver los estados del sistema pero no se evaluará ningún ítem por consola. Esto implica que hacer *print* del resultado una función, método o atributo no valida, en la corrección, que dicha función esté correctamente implementada. Todo debe verse reflejado en la interfaz. En el caso del servidor, que no tiene interfaz, el uso de *prints* es recomendado para mostrar el estado del sistema.

## 12. Restricciones y alcances

- Esta tarea es **estrictamente individual**, y está regida por el **Código de honor de Ingeniería**.
- Tu programa debe ser desarrollado en Python 3.6.
- Si no se encuentra especificado en el enunciado, asume que el uso de cualquier librería Python está prohibido. Pregunta en la *issue* especial del foro si es que es posible utilizar alguna librería en particular.
- Debes adjuntar un archivo `README.md` **conciso y claro**, donde describas los alcances de tu programa, cómo correrlo, las librerías usadas, los supuestos hechos, y las referencias a código externo. **Tendrás hasta 24 horas después del plazo de entrega** de la tarea para subir el *readme* a tu repositorio.
- Tu tarea podría sufrir los descuentos descritos en la **guía de descuentos**.
- Entregas con atraso de más de 24 horas tendrán calificación mínima (1,0).
- Cualquier aspecto no especificado queda a tu criterio, siempre que no pase por sobre otro.

Las tareas que no cumplan con las restricciones del enunciado obtendrán la calificación mínima (1,0).