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

IIC2233 Programación Avanzada (2022-1)

Tarea 3

Entrega

- **Avance de tarea**
  - **Fecha y hora:** miércoles 8 de junio de 2022, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T3/`
- **Tarea**
  - **Fecha y hora:** sábado 18 de junio de 2022, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T3/`
- `README.md`
  - **Fecha y hora:** sábado 18 de junio de 2022, 22:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T3/`

Objetivos

- Tomar decisiones de diseño y modelación en base a un documento de requisitos.
- 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 seguir un protocolo de comunicación preestablecido.

Índice

1. *DCCasillas*

Luego de haber ayudado exitosamente al *DCComando espacial* a destruir los *aliens* invasores, te enteras de que una antigua *gangster* conocida ha dado un gran anuncio: *BigCat*, quien ha conseguido fama, dinero y poder ~~enviando a los estudiantes a apostar en el *DCCasino*~~, decidió dejar su cargo para dedicarse a sus *hobbies* dando paso a una nueva era de programadores. Sus últimas palabras: ¿el *One Code*? Se los daré, pero deben ganárselo en una partida de mi brillante juego *DCCasillas*.

La noticia ha corrido rápidamente y los aspirantes comenzaron a prepararse al instante, pero solo uno puede ser el ganador y alzarse con la gloria. Los candidatos deberán luchar en una partida del apasionado *DCCasillas* para así obtener el ansiado tesoro y el título de *Rey de los Programadores*.

Ante este gran revuelo, todos tienen planeado hacer lo posible para ganar el preciado puesto, por lo que decides anticiparte y crear tu propio *DCCasillas* para entrenar antes del gran evento. Las recompensas por esto van desde una jugosa tarjeta de presentación hasta un gran vistazo al *One Code*, el cual nadie ha visto en siglos. ¡No pierdas tu oportunidad y juega por la gloria!

Figura 1: Logo de *DCCasillas*

2. Flujo del programa

DCCasillas es un juego multijugador por turnos donde cada jugador posee dos **fichas** de su **color** que deberá llevar exitosamente a su propia meta por un camino que se encuentra orientado en forma horaria. Para esto, deberán lanzar un dado y avanzar un cierto número de casillas dependiendo del resultados arrojado. El primero que logre trasladar ambas fichas a su meta será el ganador.

Además, el programa debe contar con un **servidor** funcional que permita la transmisión de mensajes entre cada jugador, de tal manera que se pueda manejar el flujo del juego. Lo primero que debe ejecutarse en tu tarea es el servidor, seguido de los clientes.

Cada jugador debe ser una instancia de la clase del cliente, la cual está encargada de la interfaz gráfica del jugador y será el único medio de comunicación con el servidor del programa. Al correr tu programa, el jugador debe poder visualizar la Ventana de inicio, donde deberá ingresar su nombre con el cual va a participar. El nombre debe tener un largo **mayor o igual a 1** carácter y **menor o igual a 10**. Tendrás que asegurarte de que sea **alfanumérico** y **no puede repetirse** entre los demás jugadores. Luego, podrás seleccionar un botón en el cual **el servidor se encargará** de verificar que tus opciones ingresadas sean válidas. Si el formato es incorrecto, se debe notificar al jugador mediante un mensaje de texto por medio

de la interfaz. En caso de cumplir con el formato solicitado, la Ventana de inicio se cerrará y se abrirá la Ventana de espera. Por el contrario, si la sala se encuentra llena o la partida está en curso, se deberá informar al usuario por medio de un mensaje en la ventana de inicio.

Una vez ingresado a la sala de espera, se te asignará un color de forma **aleatoria** y deberás visualizar a los clientes que hayan ingresado o los que vayan ingresando a la sala. El primer cliente en ingresar será el **administrador**, el cual debe ser capaz de iniciar la partida siempre que haya un `MINIMO_JUGADORES`[1] conectados, o bien apenas ingresen un `MAXIMO_JUGADORES` a la sala. Si se quisiera comenzar una partida sin cumplir con el `MINIMO_JUGADORES`, se debe informar del error en pantalla. Una vez iniciado el juego pasarán todos los jugadores a la Ventana de juego.

En la Ventana de juego se mostrará el tablero, las fichas en el lugar correspondiente, el avatar, nombre del jugador, número de turno, cantidad de fichas en la base y cantidad de fichas en la zona de color. El juego estará dado por turnos y el orden será indicado por el **mismo orden de ingreso** que el de la Ventana de espera. El resto de las reglas estará dado en la sección de Reglas de *DCCasillas*. Una vez que un jugador obtenga la victoria, se deberá desplegar la Ventana Final para cada uno de los clientes, el cual mostrará el ganador junto a las estadísticas de la partida. Finalmente, los jugadores se redirigirán a la Ventana de inicio.

3. Networking

Para poder ganar la partida de *DCCasillas*, tendrás que usar todos tus conocimientos de *networking*. Deberás desarrollar una arquitectura `cliente - servidor` con el modelo **TCP/IP** haciendo uso del módulo **socket**.

Tu misión será implementar dos programas separados, uno para el **servidor** y otro para el **cliente**. De los mencionados, **siempre** se deberá ejecutar primero el servidor y este quedará escuchando para que se puedan conectar uno o más clientes. Debes tener en consideración que la comunicación es siempre entre cliente y servidor, **nunca directamente entre clientes**.

3.1. Arquitectura `cliente - servidor`

Las siguientes consideraciones **deben ser cumplidas al pie de la letra**. Lo que no esté especificado aquí puedes implementarlo según tu criterio, siempre y cuando cumpla con lo solicitado y no contradiga nada de lo indicado (puedes **preguntar** si algo no está especificado o si no queda completamente claro).

3.1.1. Separación funcional

El cliente y el servidor deben estar separados, esto implica que deben estar en directorios diferentes, uno llamado `cliente` y otro llamado `servidor`. Cada directorio debe contar con los archivos y módulos necesarios para su correcta ejecución, asociados a los recursos que les correspondan, además de un archivo principal `main.py`, el cual inicializa cada una de estas entidades funcionales. La estructura que debes seguir se indica en el siguiente diagrama:

[1] Para esta tarea, puedes asumir que el mínimo de jugadores será mayor o igual a 2, y el máximo menor o igual a 4.

```
T3
├── cliente
│   ├── main.py
│   ├── parametros.json
│   ├── archivo_del_cliente.dcc
│   ├── sprites
│   └── ...
├── servidor
│   ├── main.py
│   ├── parametros.json
│   ├── archivo_del_servidor.dcc
│   └── ...
├── .gitignore
├── README.md
├── otro_archivo.dcc
└── ...
```

Si bien, las carpetas asociadas al `cliente` y al `servidor` se ubican en el mismo directorio (`T3`), la ejecución del `cliente` **no debe depender de archivos en la carpeta del `servidor`**, y la ejecución del `servidor` **no debe depender de archivos y/o recursos en la carpeta del `cliente`**. Esto significa que debes tratarlos como si estuvieran ejecutándose en computadores diferentes. La figura 2 muestra una representación esperada para los distintos componentes del programa:

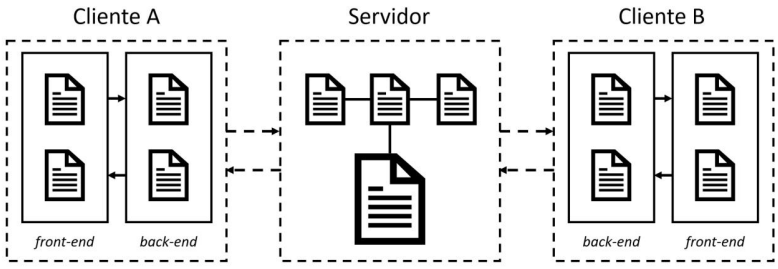


Figura 2: Separación cliente-servidor y *front-end-back-end*.

Cabe destacar que **solo el cliente tendrá una interfaz gráfica**. Por lo tanto, todo cliente debe contar con una separación entre *back-end* y *front-end*, mientras que la comunicación entre el `cliente` y `servidor` debe realizarse mediante *sockets*. Ten en cuenta que la separación deberá ser de caracter **funcional**. Esto quiere decir que **toda tarea** encargada de **validar y verificar acciones** deberá ser **elaborada en el servidor**, mientras que el cliente deberá solamente recibir esta información y actualizar la interfaz.

3.1.2. Conexión

El `servidor` contará con un archivo de formato `JSON`, ubicado en la carpeta del `servidor`. Este archivo debe contener los datos necesarios para instanciar un *socket*. El archivo debe llevar el siguiente formato:

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

Por otra parte, el `cliente` deberá conectarse al *socket* abierto por el `servidor` haciendo uso de los datos encontrados en el archivo `JSON` de la carpeta del `cliente`. Es importante recalcar que el `cliente` y el `servidor` **no deben usar el mismo archivo** `JSON` para obtener estos parámetros.

3.1.3. Método de codificación

Cuando se establece la conexión entre el `cliente` y el `servidor`, deberán encargarse de intercambiar información constantemente entre sí. Por ejemplo, el `cliente` le comunica al `servidor` la cantidad de espacios que debe avanzar su ficha y este le responderá con un mensaje indicando la nueva posición de esta. ¡Pero cuidado! Como no queremos que un contrincante pueda *hackear* nuestro mensaje y avanzar más de lo que debería, debes asegurarte de encriptar el contenido antes de enviarlo, de tal forma que si alguien lo intercepta no pueda descifrarlo. El método de encriptación se explicará en detalle más adelante. **Luego** de encriptar el contenido, deberás codificar los mensajes enviados entre el `cliente` y el `servidor` según la siguiente estructura:

- Debes dividir el mensaje en **bloques**, cuya estructura será especificada un poco más adelante.
- Los primeros 4 *bytes* del mensaje deben indicar **la cantidad de bloques** que se enviarán (es decir, la cantidad de bloques en la que se dividió el contenido del mensaje). Estos 4 bytes deben estar en formato *little endian*.[2]
- A continuación, debes enviar el contenido encriptado del mensaje. Este debe separarse en bloques de 22 *bytes*, en donde solo 20 serán ocupados por el contenido del mensaje. Cada bloque debe estar precedido de 4 *bytes* que indiquen el número del bloque, comenzando a contar desde cero y codificados en *big endian*. Si para el último bloque no alcanza el mensaje para completar los 20 *bytes*, deberás hacer este bloque más pequeño que los demás. El contenido debería verse algo así:

| 4 bytes | **4bytes** | bloque1 | **4bytes** | bloque2 | **4bytes** | bloque3 | ... |

Cuadro 1: Método de codificación

- Finalmente, cada bloque debe estar compuesto por 22 *bytes*, como se mencionó anteriormente. De estos, el primero representará si se utilizó la totalidad de los 20 *bytes* reservados para el mensaje o no, teniendo un valor de uno (`b'\x 01'`) en caso de que si y un valor de cero (`b'\x 00'`) en caso de que no. El segundo *byte* debe indicar la cantidad de *bytes* utilizados por el mensaje en el bloque. Finalmente, los 20 bytes restantes deben ser utilizados (si es que son ocupados en su totalidad) por el contenido encriptado del mensaje. El contenido de cada bloque debería verse algo así:

| 1 byte + 1 byte + min(largo, 20 bytes) |

Cuadro 2: Estructura de cada bloque

3.1.4. Método de encriptación

Finalmente, el **método de encriptación** que deberás realizar para esconder el contenido de tus mensajes y evitar *hackeos* es el siguiente:

- El mensaje deberá ser separado en **dos partes**. Cada parte es construida de la siguiente manera:
  1. Se agrega a la **primera** parte el primer byte.

[2] El *endianness* es el orden en el que se guardan los bytes en un respectivo espacio de memoria. Esto es relevante porque cuando intentes usar los métodos `int.from_bytes` e `int.to_bytes` deberás proporcionar el *endianness* que quieras usar, además de la cantidad de bytes que quieres usar para representarlo. Para más información puedes revisar este **enlace**.

2. Se agrega a la **segunda** parte el byte que le sigue al anterior.
3. Se agregan a la **primera** parte los 2 bytes siguientes al anterior.
4. Se agregan a la **segunda** parte los 2 bytes siguientes al último anterior.
5. Se repite el proceso hasta agregar el último byte del mensaje.

Llamemos A y B a las 2 partes por construir. Para construir A se debe usar el algoritmo descrito, comenzando por el primer *byte* del mensaje, mientras que para construir B se debe usar el mismo algoritmo, pero comenzando desde el segundo *byte* del mensaje. Por ejemplo, sea X la secuencia de *bytes* a encriptar, las partes A y B quedan así:

$$X = b_0\ b_1\ b_2\ b_3\ b_4\ b_5\ b_6\ b_7\ b_8\ b_9\ b_{10}$$
$$A = b_0\ b_2\ b_3\ b_6\ b_8\ b_9$$
$$B = b_1\ b_4\ b_5\ b_7\ b_{10}$$

Notar que A y B no tienen por que tener el mismo largo.

- Luego, se analiza la suma de los valores numéricos de los *bytes* centrales de cada una de las partes. Para saber cuales son los *bytes* centrales y su suma, usemos el ejemplo anterior: En el caso de A, que tiene un número par de *bytes*, se consideran como centrales los *bytes* centrales como b_3 y b_6. En el caso de B, los bytes centrales serán b_4, b_5 y b_7. Por otro lado, los valores que se compararán serán, en el caso de A, la suma de los valores de los *bytes* centrales. Para el caso de B, se sumará el valor del *byte* más central (b_5) con el promedio de los valores de los dos restantes (b_4 y b_7).

  - Si la suma de los valores centrales de A es menor o igual a la suma de los valores centrales de B, el resultado encriptado será
  $$nAB$$
  donde n es igual a 1 para indicar que se cumplió esta condición.
  - Si la suma de los valores centrales de A es mayor a la suma de los valores centrales de B, el resultado encriptado será
  $$nBA$$
  donde n es igual a 0 para indicar que se cumplió esta condición.

  Ahora, lo último que necesitarás para completar la comunicación segura es **diseñar una forma para desencriptar el mensaje** desde el *receptor* (`cliente` o `servidor`).

3.1.5. Ejemplo de encriptación

Para ayudarte a entender de mejor forma el método de encriptación que **debes** implementar, utilizaremos como ejemplo el siguiente contenido de mensaje (previo a la encriptación):

b'\x05\x08\x03\x02\x04\x03\x05\x09\x04'

Lo primero que hacemos es separar el ***bytearray*** en 2 partes de la siguiente forma:

A = b'\x05\x03\x02\x05\x04'
B = b'\x08\x04\x03\x09'

Por lo tanto, como el número de *bytes* de A es **impar**, entonces sus *bytes* centrales de A están dados por \x03\x02\x05. Por lo tanto, la suma de sus *bytes* centrales será:

$$2 + \frac{3+5}{2} = 6$$

Mientras que B, al ser **par**, la suma de sus *bytes* centrales estará dado por la suma de \x04\x03:

$$4 + 3 = 7$$

Como la suma de los *bytes* centrales de A es menor o igual a la de B, entonces el resultado estará dado por el formato nAB. Es decir, insertamos un 1, luego A y finalmente B, quedando como resultado final de la encriptación:

b'\x01\x05\x03\x02\x05\x04\x08\x04\x03\x09'

3.1.6. Ejemplo de codificación

Para ayudarte a entender el método de codificación explicado anteriormente, supongamos que se quiere enviar el mensaje ya encriptado mostrado en el ejemplo anterior desde el `servidor` al `cliente`. Lo primero, es identificar en cuántos bloques se enviará el mensaje y, como podemos ver, el largo de este es de solo 10 bytes, por lo que se enviará solo un bloque con el contenido de este. Luego, como fue explicado, necesitamos anteponer 2 bytes al contenido del mensaje dentro del bloque. Como no se usó la totalidad de los 20 bytes del bloque, el primero de estos bytes será un \x00 y el segundo será un \n. [3] (equivalente al valor de 10 *bytes*). Finalmente, al bloque debemos anteponerle el número de bloque en el formato especificado y a todo esto anteponerle la cantidad de bloques usados. La estructura debería verse así:

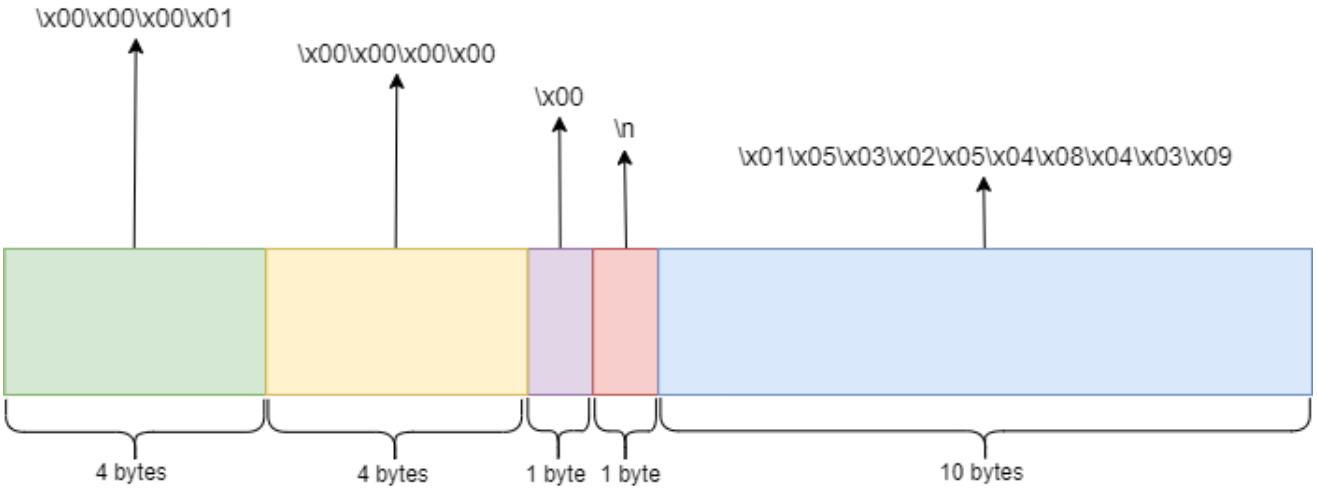


Figura 3: Ejemplo de estructura del *bytearray* para un mensaje codificado

3.1.7. *Logs* del servidor

Como el servidor no cuenta con interfaz gráfica, debes indicar constantemente lo que ocurre en él mediante la consola. Estos mensajes se conocen como mensajes de *log*, es decir, deberás llamar a la función `print` cuando ocurra un evento importante en el servidor. Estos mensajes se deben mostrar para cada uno de los siguientes eventos:

[3] Recuerda que b'\x0A' es equivalente a b'\n'.

- Se conecta un cliente al servidor
- Cuando se comienza una partida. Debe indicar quienes están en esa partida.
- Cuando un jugador avanza en el tablero. Debe indicar quien y cuanto avanzó.
- Cuando una ficha come a otra. Debe indicar de quien era cada ficha.
- Un cliente ingresa un nombre de usuario. Debes indicar el nombre y si es válido o no.
- Comienza el turno de un cliente. Debes indicar su nombre.
- Se termina la partida, debes indicar cual fue el ganador de la partida.

Es de **libre elección** la forma en que representes los *logs* en la consola, un ejemplo de esto es el siguiente formato:

```
     Cliente     |        Evento          |      Detalles
-----------------|------------------------|-----------------------
     drcid98     |       Conectarse       |          -
-----------------|------------------------|-----------------------
    gvfigueroa   |  Se movió X espacios   |          -
-----------------|------------------------|-----------------------
        -        |   Término de partida   | Ganador: matiasmasjuan
-----------------|------------------------|-----------------------
```

3.1.8. Desconexión repentina

En caso de que algún ente se desconecte, ya sea por error o a la fuerza, tu programa debe reaccionar para resolverlo:

- Si es el `servidor` quien se desconecta, cada cliente conectado debe mostrar un mensaje explicando la situación antes de cerrar el programa.
- Si es un `cliente` quien se desconecta, se descarta su conexión. Si se desconecta mientras está en una partida, entonces deja de ser considerado en la lista de jugadores y siguen jugando los que queden en sala. En caso de que al desconectarse deje a su oponente solo, este ganará automáticamente la partida.

3.2. Roles

A continuación se detallan las funcionalidades que deben ser manejadas por el servidor y las que deben ser manejadas por el cliente.

3.2.1. Servidor

- **Procesar y validar** las acciones realizadas por los clientes. Por ejemplo, si deseamos comprobar que el jugador ingresa un nombre correcto, el servidor es quien debe verificarlo y enviarle una respuesta al cliente según corresponda.
- **Distribuir y actualizar** en tiempo real los cambios correspondientes a cada uno de los participantes del juego, con el fin de mantener sus interfaces actualizadas y que puedan reaccionar de manera adecuada. Por ejemplo, si un jugador mueve una ficha en una partida, el servidor deberá notificar de esto a todos los clientes en la partida y se deberá reflejar en la interfaz de cada uno de ellos.

- **Almacenar y actualizar** la información de cada cliente y sus recursos. Por ejemplo, el servidor manejará la información de la posición de cada ficha que tiene cada uno de los clientes conectados y las actualizará una vez que avancen.

3.2.2. Cliente

Todos los clientes cuentan con una interfaz gráfica que le permitirá interactuar con el programa y enviar mensajes al servidor. Maneja las siguientes acciones:

- **Enviar todas las acciones** realizadas por el usuario hacia el servidor.
- **Recibir e interpretar** las respuestas y actualizaciones que envía el servidor.
- **Actualizar** la interfaz gráfica de acuerdo a las respuesta que recibe del servidor.

4. Reglas de *DCCasillas*

4.1. Preparación

Antes de comenzar el juego, todos los jugadores se encontrarán en la Ventana de espera, en donde se les otorgará un color aleatorio a cada jugador a medida que vayan ingresando a la sala. El primer jugador en ingresar a esta sala será nombrado como **administrador** de la partida, y se le dará la opción de pulsar el botón de **empezar partida** en cualquier momento, siempre y cuando haya un mínimo de `MINIMO_JUGADORES`[4] jugadores en la sala. En caso de que no se cumpla esta condición, se le avisará al administrador que aún no es posible comenzar la partida. Si el administrador no pulsa el botón, la partida iniciará automáticamente cuando el número de jugadores en la sala llegue a `MAXIMO_JUGADORES`.

En caso de que un jugador quisiera unirse a la partida, vale decir, entrar a la sala de espera desde la ventana de inicio, pero esta se encuentre llena o ya haya iniciado, se le avisará en la misma ventana inicial.

4.2. Juego

DCCasillas consiste en un juego de estrategia basado en turnos, en que los jugadores compiten por hacer que sus fichas lleguen a la meta antes que las del resto. Para esto, los jugadores deben lanzar un **dado de tres caras**. El primer jugador que logre llevar sus dos fichas a la meta será el ganador.

El juego cuenta con un sistema de turnos dado por el orden de llegada a la **Sala de Espera**, por ende, el primer jugador en llegar a esta sala cuenta con el primer turno de la partida, el segundo en llegar con el segundo turno, y así para todos los jugadores conectados.

El **tablero** de juego está compuesto por una grilla de tamaño 6x6. Cada esquina tendrá un color asociado, las cuales al iniciar el juego comenzarán con dos fichas, indicando las fichas iniciales de cada jugador. El movimiento de las fichas se realizará en sentido **horario** y la cantidad de casillas que avanzarán estará dado por el número arrojado por el lanzamiento del dado del jugador, cuyo valor será un número **aleatorio** entre **1** y **3**. Una vez que la ficha avance las casillas que le corresponda, se pasará el turno al siguiente jugador.

[4] Para esta tarea, puedes asumir que el mínimo de jugadores será mayor o igual a 2, y el máximo menor o igual a 4.

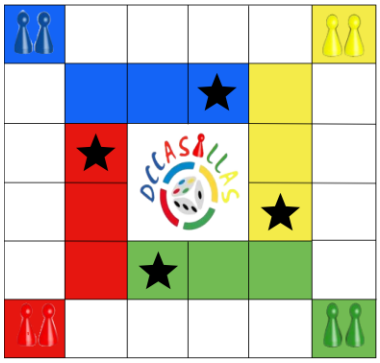


Figura 4: Ejemplo del tablero inicial

El movimiento de las fichas se realizará **solo a través** de las casillas **blancas** y las casillas de la **zona de color**. Cada ficha tendrá como objetivo dar una vuelta en sentido **horario** pasando por todas las casillas blancas, pero antes de llegar a su casilla de origen, deberán seguir la ruta de su **derecha** e ingresar a la **zona de color**. Una vez dentro de esta sección del tablero, solo podrás llegar a la meta (indicada por la estrella) si arrojas un número **exacto** para llegar a ella, o un número menor. Por ejemplo, si estás a 2 casillas de la meta y te sale un 3, no te podrás mover y perderás un turno. En cambio, si estás a 2 casillas de la meta y arrojas un 1, te deberás mover una casilla y por consecuencia el número exacto que te faltará para llegar a la meta con esa ficha será 1 casilla. Una vez que la primera ficha de un jugador llega a la meta, este podrá jugar con su segunda ficha desde su próximo turno. Cuando un jugador lleva su segunda ficha a la meta, este ganará y se terminará la partida.

Un evento que puede suceder durante el transcurso del juego es la superposición de fichas. Si la ficha del jugador del turno cae en la misma casilla en que se encontraba la ficha de algún otro jugador, la ficha de este último (del jugador que estaba primero en la casilla) deberá volver a la casilla de inicio de su color y deberá comenzar el camino a la meta desde cero. A este evento se le conoce como **comer una ficha**.

4.3. Ejemplos de movimiento

4.3.1. Comer una ficha

Supongamos que tenemos la configuración del tablero de la izquierda. Además, es el turno del jugador **rojo** y obtiene un **2** en el lanzamiento del dado. Como el movimiento es horario, deberá ir hacia arriba y luego a la derecha cruzando solamente por las casillas **blancas**. Primero sube una casilla, y al subir la segunda está cruzando la casilla inicial de los azules, por lo que debe moverse a la derecha. Sin embargo, como en la casilla resultante está posicionado una ficha **azul**, entonces la ficha **roja** se **comerá** a la **azul** y esta volverá a su inicio. La imagen de la derecha muestra como quedará el tablero tras este movimiento.

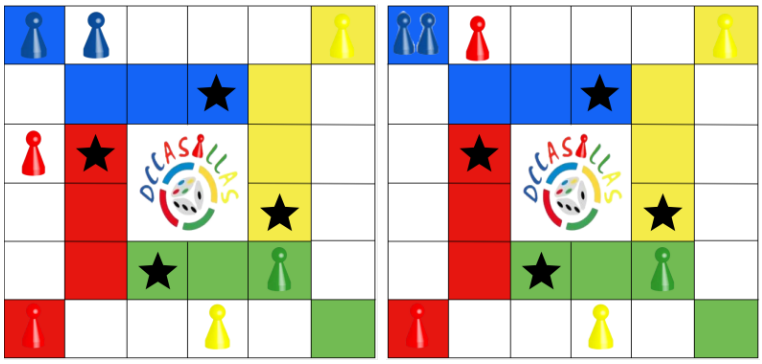


Figura 5: Ejemplo de movimiento al comer una ficha

4.3.2. Entrar a zona de color

Supongamos que tenemos la configuración del tablero de la izquierda. Además, es el turno del jugador **rojo** y obtiene un **3** en el lanzamiento del dado. Como el movimiento es horario, deberá ir hacia la izquierda cruzando solamente por las casillas **blancas**. Primero avanza una casilla, pasando por la ficha amarilla, pero al no terminar en esa casilla entonces la pasa, sin devolverla a su casilla original. Luego avanza otra a la izquierda, y antes de volver a avanzar, deberá **girar** a su derecha y avanzar una casilla para entrar a la **zona de color**. La imagen de la derecha mostraría el resultado final luego del movimiento.

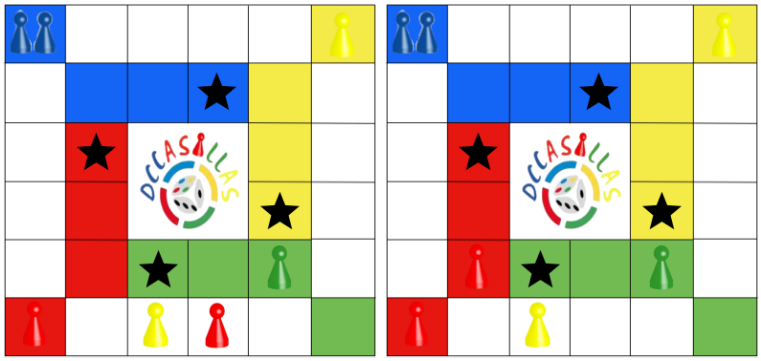


Figura 6: Ejemplo de movimiento al entrar a zona de color

4.4. Fin del juego

Una vez finalizado el juego se debe desplegar la Ventana Final, la cual mostrará quien es el ganador y un resumen de las estadísticas del estado final de los demás jugadores, señalando su cantidad de fichas en la casilla de inicio, las fichas en su zona de color y las fichas en la meta. Luego deberá devolver a todos los jugadores a la **Ventana de inicio**.

5. Interfaz gráfica

5.1. Ventana de inicio

Esta ventana se despliega al inicio del programa. Debe permitir que se introduzca el nombre del jugador, que será con el que se identificará y que se mostrará a los demás jugadores de la partida. El nombre debe cumplir con los mismos requisitos indicados en Flujo del programa. Si el nombre es válido, se continúa a la Sala de espera, mientras que en caso contrario se debe avisar al usuario y permitir ingresar otro nombre.

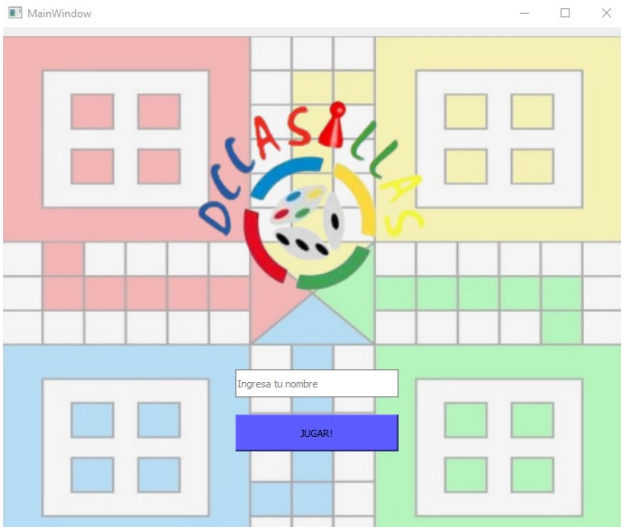


Figura 7: Ventana de inicio

En el caso de que un usuario se conecte y la sala de espera este llena o la partida ya haya comenzado, se le debe notificar y mantener en la ventana de inicio

5.2. Ventana de espera

Esta ventana se despliega luego de introducir el nombre en el inicio. En esta se despliegan los nombres de los jugadores que se están uniendo para jugar en la partida. Junto con los nombres de los jugadores se debe mostrar el color aleatorio y el turno que se le asignó a cada uno al ingresar a la sala de espera.

El primer jugador en ingresar a la sala, será el **administrador**, quien tiene la facultad de iniciar la partida cuando quiera, siempre y cuando haya como mínimo `MINIMO_JUGADORES`. Para esto tendrá un botón que le permite iniciar la partida.

En el caso de que en la sala de espera se alcance un `MAXIMO_JUGADORES`, y el administrador no haya presionado el botón de iniciar partida, esta se iniciará automáticamente. Luego de iniciar partida, todos los jugadores avanzan a la ventana de juego.

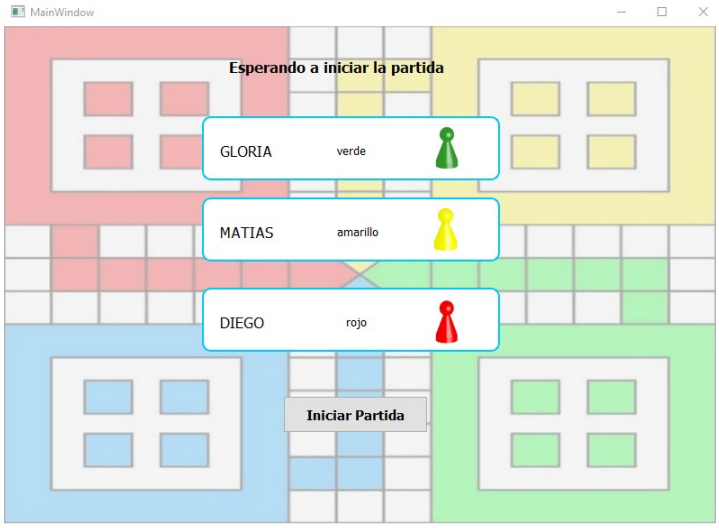


Figura 8: Ventana de espera del administrador

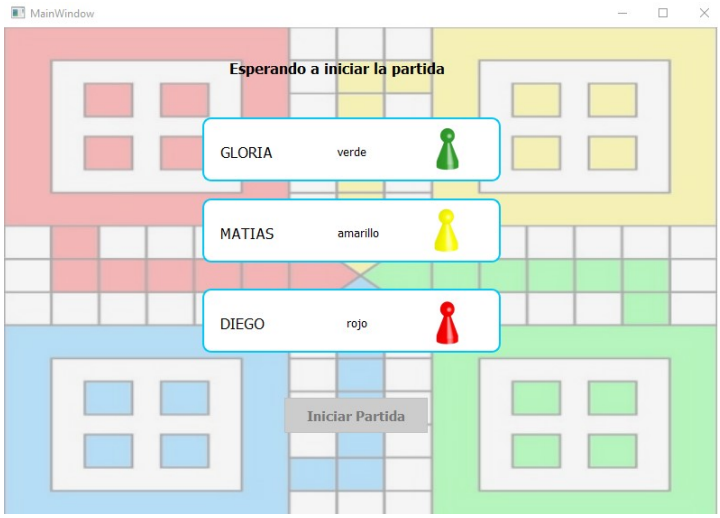


Figura 9: Ventana de espera jugador normal

5.3. Ventana de juego

En esta sala es donde se desarrollará la partida. Debe tener:

- El tablero por donde se moverán las fichas de los jugadores
- Un botón para lanzar el dado, encargado de pedirle al servidor que procese el lanzamiento del dado del jugador.
- Un resumen de cada uno de los jugadores, donde se muestre el usuario del jugador, el avatar asociado, el turno asignado de acuerdo al orden de llegada, la cantidad de fichas que tiene en la base, la cantidad de fichas que tiene en a la zona de color tras dar la vuelta completa, y la cantidad de de fichas que ya llegaron a la victoria, es decir la ultima casilla de su respectivo color.
- Una sección donde se indique quien está jugando su turno actualmente
- Una estrella en la celda de victoria de cada color
- Las fichas de los jugadores que se mueven en sentido horario dependiendo del resultado del dado y del jugador que este en su turno

Figura 10: Ventana de juego

5.4. Ventana Final

Esta ventana muestra al jugador ganador. Y se muestra el estado con el que terminaron los demás jugadores, mostrando, para cada uno, la cantidad de fichas que quedaron en la base, la cantidad de fichas en la zona de color y la cantidad de fichas en la zona de victoria. Finalmente devuelve a todos los jugadores a la ventana de inicio

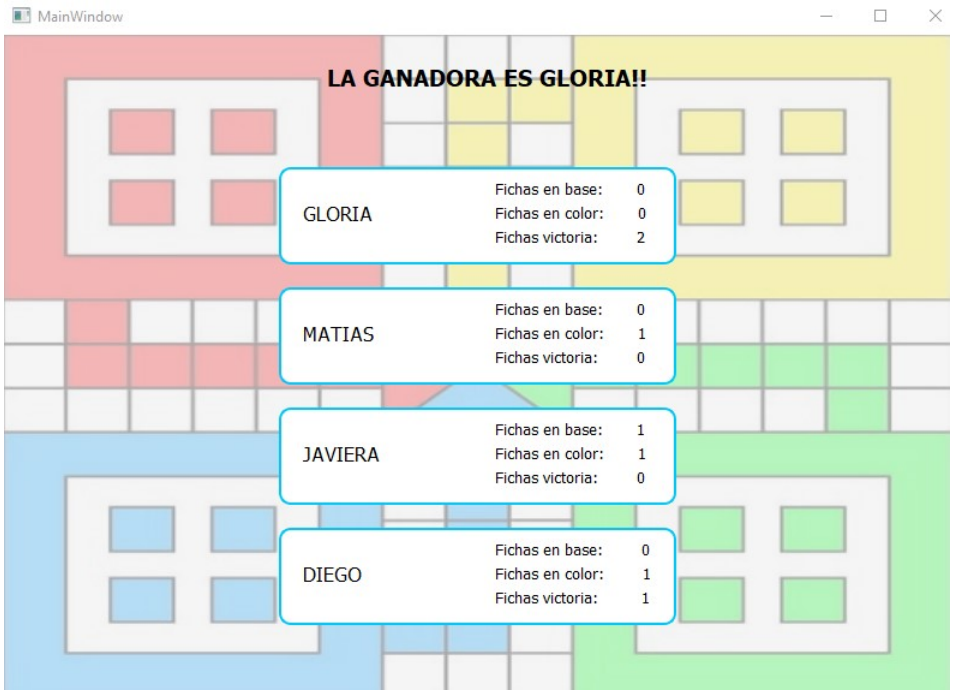


Figura 11: Ventana final

6. Archivos

Para desarrollar tu programa de manera correcta, deberás crear o utilizar los siguientes archivos:

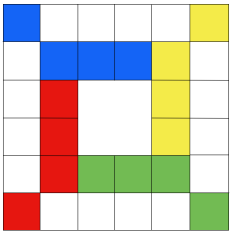
6.1. Archivos entregados

6.1.1. Sprites

Carpeta que contiene todos los elementos visuales que necesitas para llevar a cabo tu tarea, entre ellos encontrarás las subcarpetas:

- **Fichas:** Contiene la subcarpeta llamada *Simples* donde se encuentran las fichas individuales que se mueven a lo largo del tablero. Están en formato ficha-x, siendo x el color del jugador correspondiente. También contiene la subcarpeta llamada *Dobles* donde se encuentran los sprites de los pares de fichas que inician en la base de cada jugador y están en formato fichas-x siendo x el color del jugador correspondiente.
- **Juego:** Contiene el tablero en donde se desarrollará la partida y las estrellas que van ubicadas en la zona de victoria (ultima casilla de color) de cada jugador.
- **Logos:** Contiene el logo del juego, un fondo de un tablero de *DCCasillas* y un dado, para que decoren sus ventanas como más les guste.

(a) *Sprite* para ficha



(b) *Sprite* para tablero

Figura 12: Ejemplos de *sprites* para cada carpeta

6.2. Archivos a crear

6.2.1. `parametros.json`

Dentro de tu tarea deben existir dos archivos de parámetros independientes entre sí: uno para el servidor, y otro para el cliente. Cada archivo de parámetros debe contener solamente los parámetros que corresponden a su respectiva parte, es decir, `parametros.json` en la carpeta `cliente/` deberá contener solamente parámetros útiles para el cliente (como los *paths*), mientras que `parametros.json` de `servidor/` contendrá solamente parámetros útiles para el servidor. Los archivos deben encontrarse en formato `JSON`, como se ilustra a continuación:

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

7. *Bonus*

En esta tarea habrá una serie de *bonus* que podrás obtener. Cabe recalcar que necesitas cumplir los siguientes requerimientos para poder obtener *bonus*:

1. La nota en tu tarea (sin bonus) debe ser **igual o superior a 4.0**[5].
2. El bonus debe estar implementado **en su totalidad**, es decir, **no se dará puntaje intermedio**.

Finalmente, la cantidad máxima de décimas de *bonus* que se podrá obtener serán 5 décimas. Deberás indicar en tu `README` si implementaste alguno de los bonus, y cuáles fueron implementados.

7.1. Bonus Cheatcode (1 décima)

Si notas que la partida no va a tu favor, puedes perjudicar a tus contrincantes implementando **un cheatcode**:

- **W + P**: Al presionar esta combinación de teclas, las fichas de los todos los jugadores (**exceptuando** el tuyo) volverán a sus casillas de inicio.

7.2. Turnos con Tiempo (2 décimas)

La serie favorita de *BigCat* ya va a comenzar y las partidas están durando demasiado. Enojada e impaciente te pide que programes un reloj para que el ganador se decida dentro de un límite de tiempo y así no toparse con ningún *spoiler*.

Para obtener el puntaje de este bonus deberás implementar un *timer* de `TIEMPO_TURNO` segundos que se inicie automáticamente al comenzar el turno de algún jugador. Este debe ser visible por todos los jugadores, por lo que el tiempo restante se debe transmitir al resto de los clientes dentro de la partida. Si el jugador no lanza el dado antes de que acabe el tiempo termine, no podrá mover sus fichas y se pasará de turno al siguiente jugador, reiniciándose el tiempo.

7.3. Rebote (2 décimas)

Aburrido de estar dentro de la zona de color y no poder moverte ya que te salen números mayores a la cantidad de casillas que te faltan para llegar a la meta, decides implementar el efecto rebote de tal forma que independiente del número que te salga en el dado, puedas mover la ficha.

Una vez llegada a la zona de color, si obtienes un número que excede al necesario para llegar a la meta, deberás avanzar la cantidad de casillas hasta ella y luego retroceder la cantidad restante, simulando el efecto de rebote. Por ejemplo si estás a 2 casillas de la meta y te obtienes un 3, deberás avanzar las dos casillas hasta la meta y luego retroceder el restante, es decir, una casilla.

8. Avance de tarea

Para esta tarea, el avance corresponde a implementar el **inicio de la interacción con** *DCCasillas*, con un servidor y un cliente implementado.

En específico, para este avance deberás entregar un servidor capaz de conectarse a múltiples clientes, que sea capaz de recibir nombres para ingresar a la sala de espera y les permita ingresar solo si el nombre cumple las condiciones pedidas en el Flujo del programa. También deberás entregar un cliente que tenga implementadas la Ventana de inicio y la Ventana de espera, de modo tal que el usuario pueda interactuar

[5] Esta nota es sin considerar posibles descuentos.

con el servidor y tener acceso a la ventana de espera desde la ventana de inicio siempre y cuando cumpla con las condiciones anteriores.

Para fines de este avance, no es necesario implementar la actualización en vivo de los clientes que irán ingresando a la Ventana de espera, ni una distinción visual en caso de ser **administrador** o no. Sin embargo, es necesario que se realice como **mínimo** el protocolo de envío de mensajes, considerando tanto la **codificación** como la **decodificación**. No es necesario implementar la **encriptación**. Quedará a tu criterio si decides implementarlo o no para este avance.

A partir de los avances entregados, se les brindará un *feedback* general de lo que implementaron en sus programas y además, les permitirá optar por **hasta 2 décimas** adicionales en la nota final de su tarea.

9. `.gitignore`

Para esta tarea **deberás utilizar un** `.gitignore` para ignorar los archivos indicados, este deberá estar dentro de tu carpeta `Tareas/T3/`. Puedes encontrar un ejemplo de `.gitignore` en el siguiente ***link***.

Para esta tarea, deberás ignorar en específico los siguientes archivos:

- Enunciados
- `Sprites`

Se espera que no se suban archivos autogenerados por las interfaces de desarrollo o los 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.

10. Entregas atrasadas

Posterior a la fecha de entrega de la tarea se abrirá un formulario de Google Form, en caso de que desees que se corrija un *commit* posterior al recolectado, deberás señalar el nuevo *commit* en el *form*.

El plazo para rellenar el *form* será de 24 horas, en caso de que no lo contestes en dicho plazo, se procederá a corregir el *commit* recolectado.

11. Importante: Corrección de la tarea

Para esta tarea, el carácter funcional del programa será el pilar de la corrección, es decir, **sólo 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.

Cuando se publique la distribución de puntajes, se señalará con color **amarillo** cada ítem que será evaluado a nivel de código, todo aquel que no esté pintado de amarillo significa que será evaluado si y sólo si se puede probar con la ejecución de su tarea.

En tu archivo `README.md` deberás señalar el archivo y la línea donde se encuentran definidas las funciones o clases relacionados a esos ítems.

Finalmente, si durante la realización de tu tarea se te presenta algún problema o situación que pueda afectar tu rendimiento, no dudes en contactar al ayudante jefe de Bienestar al siguiente correo: **bienestar.iic2233@ing.puc.cl**.

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 3.8.
- Tu programa debe estar compuesto por uno o más archivos de extensión `.py`.
- Si no se encuentra especificado en el enunciado, supón 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 2 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).