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

IIC2233 Programación Avanzada (2021-2)

Tarea 3

Entrega

- **Avance de tarea**
  - **Fecha y hora:** miércoles 1 de diciembre de 2021, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T3/`
- **Tarea**
  - **Fecha y hora:** sábado 11 de diciembre de 2021, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T3/`
- `README.md`
  - **Fecha y hora:** sábado 11 de diciembre de 2021, 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. *DCCalamar*

Dado el reciente incremento de estudiantes al campus San Joaquín, fue descubierto el nuevo sitio favorito de comida rápida: Pizza Planeta, por lo que varios miembros de la universidad han gastado todo su dinero en las adictivas pizzas del local. Como consecuencia de esto, los problemas financieros no tardaron en llegar. Después de días sin poder gozar de los servicios de Pizza Planeta, decides pasar las penas con un confiable handroll de luca. Por suerte (o por desgracia), dentro del handroll venía una sospechosa invitación para participar en los asombrosos juegos de *DCCalamar*, donde podrás ganar lo equivalente a 10 millones de pizzas si logras pasar la prueba presentada. Claramente, al ser una propuesta tan tentadora y segura, decides marcar al número de la tarjeta e inscribirte de inmediato.

En esta llamada, te informan que tienes 2 semanas para prepararte para el *DCCalamar*, el cual consta de un sólo juego usando canicas, donde tu participación consiste en competir contra el resto de jugadores, de manera que se generará un combate de dos jugadores. Los combates continúan hasta que sólo uno quede *vivo*. Si bien las instrucciones de la modalidad del juego parecen simples, siempre habrá un guardia con una metralleta gigante vigilando tu fracaso o derrota en el juego.

Considerando tu valiosa labor como programador, decides gastar estas 2 semanas en programar una simulación del juego que te describieron, con el objetivo de poder practicar todo lo que puedas para poder ser el ganador del *DCCalamar* y poder saciar tu hambre de pizzas en todo momento.

Figura 1: Logo de *DCCalamar*

2. Flujo del programa

DCCalamar es un juego de multijugador de supervivencia donde los usuarios participarán por turnos hasta que sólo quede un único jugador, el cual será el que haya ganado en todas las competencias contra el resto. El juego consta en adivinar si la cantidad de canicas que tiene escondido el contrincante es par o impar, donde cada vez que tengas éxito en tu predicción, te quedas con todas las canicas utilizadas para ese turno. En caso de que falles, pierdes tus canicas. El jugador que consiga todas las canicas del oponente será el ganador.

DCCalamar 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 programa. 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, clase encargada de la interfaz gráfica del jugador y único medio de comunicación con el servidor del programa. Al correr tu programa, el jugador debe poder visualizar la Ventana de inicio , correspondiente a la tarjeta de invitación al juego, donde debe ingresar su nombre y la fecha de nacimiento que le permitirá participar. El nombre debe tener un **largo mayor o igual a 1** caracter **alfanumérico** y **no puede repetirse entre los jugadores**, mientras que el cumpleaños debe seguir el formato de dd/mm/yyyy. Luego, podrás seleccionar un botón que 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 y permitir el nuevo ingreso de información a los campos de nombre y cumpleaños. En caso de cumplir con el formato solicitado, la Ventana de inicio se cerrará y se abrirá la ventana de la

Sala principal, donde se verán todos los clientes conectados hasta el momento. Entre los jugadores, se deben formar parejas mediante el envío de una solicitud al jugador con el que te interesa participar. Las parejas que estén confirmadas podrán pasar a la Sala de juego.

En la sala de juego, comienza la partida en una ventana donde se muestran la pareja de jugadores, la cantidad inicial de canicas de cada uno y las condiciones para apostar tus canicas. Esta ventana debe aparecer sólo para la pareja, por lo cual si otro par de jugadores ingresa a la Sala de juego, se deberá visualizar otra ventana correspondiente a esos jugadores, respetando el nombre y avatar de cada cliente.

El juego es por turnos. El primer turno se asignará al azar a uno de los jugadores. Luego de que alguno pierda todas sus canicas, se desplegará automáticamente la Ventana final, en la cual se muestra al ganador y al perdedor del *DCCalamar*. Aquí debe estar la opción de volver a jugar, la cual dirigirá a cada jugador a la Ventana de inicio para ingresar los datos y elegir nuevamente una pareja para competir.

3. Networking

Para poder *sobrevivir* y ganar el *DCCalamar*, es necesario que pongas a prueba tu expertiz sobre *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**, donde el primero en ejecutarse **siempre** será el servidor. Luego de comenzar, el servidor queda escuchando para que uno o más clientes se conecten a él. Es importante notar que se conectan al 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:

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

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:

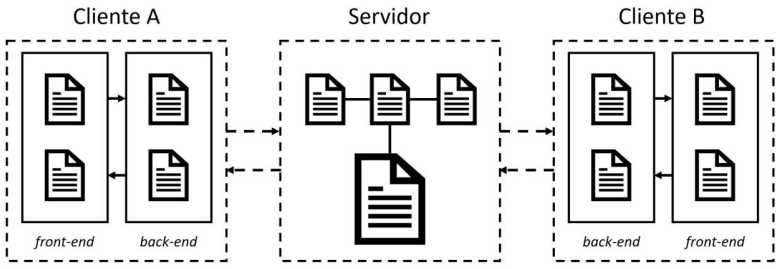


Figura 2: Separación cliente-servidor y *front-end-back-end*.

Solo el cliente tendrá una interfaz gráfica. Por tanto, un 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`, el cual contiene 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`.

3.1.3. Envío de información

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 canicas que el usuario predice que tendrá su contrincante y le responderá con un mensaje de éxito o error. ¡Pero cuidado! Como no queremos que el *enemigo* pueda *hackear* nuestro mensaje, 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:

- Los primeros 4 *bytes* deben indicar el largo del contenido encriptado del mensaje, los que tendrán que estar en formato *little endian*.[1] Por contenido se entiende al mensaje inicial que debe ser enviado.
- A continuación debes enviar el contenido encriptado del mensaje enviado. Este debe separarse en bloques de 80 *bytes*. Cada bloque debe estar precedido de 4 *bytes* que indiquen el número del bloque, comenzando a contar desde **0**, codificados en *big endian*. Si el último bloque es menor a 80 *bytes*, deberás rellenar al final con *bytes* 0 (`b'\x 00'`), hasta completar esa cantidad.
- Si deseas transformar *strings* a *bytes* deberás utilizar `UTF-8`, así no tendrás problemas con las tildes ni caracteres especiales de ningún tipo.

Finalmente, el **método de encriptación** que se te ocurrió para esconder el contenido de tus mensajes y evitar *hackeos* **es** como sigue:

- El mensaje se deberá separar en **tres partes**, donde la primera se constituye de todos los *bytes* que se encuentran a tres posiciones de distancia, comenzando desde el índice *cero*. La segunda lo mismo pero comenzando desde el *uno*, y la tercera comenzando desde el *dos*. Por ejemplo, sea X la secuencia de *bytes*, las tres partes A, B y C quedan así:

$$X = b_0\ b_1\ b_2\ b_3\ b_4\ b_5\ b_6$$
$$A = b_0\ b_3\ b_6$$
$$B = b_1\ b_4$$
$$C = b_2\ b_5$$

  Notar que la suma entre el largo de A, B y C siempre es igual al largo de X.

- Luego, se revisa si el primer *byte* de B es mayor al primer *byte* de C:
  - Si se cumple, entonces se deben intercambiar todos los 3 por 5 en la secuencia, y viceversa. Finalmente, el resultado encriptado a retornar será volver a juntar los *bytes* en el orden

$$A\ B\ C\ n$$

    donde n es igual a 0 para indicar que la condición se cumplió.

[1] 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**.

- Si no se cumple, entonces el resultado encriptado a retornar será

$$B\ A\ C\ n$$

donde n es igual a 1 para indicar que la condición no se cumplió.

Ahora, lo último que necesitarás para completar la comunicación segura es **diseñar una forma para desencriptar el mensaje** desde el *receptor* (`cliente` o `servidor`).[2]

3.1.4. 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'

Lo primero que hacemos es separar el *bytearray* en 3 partes de la siguiente forma:

A = b'\x05\x02\x05'
B = b'\x08\x04\t' [3]
C = b'\x03\x03'

Ahora, notamos que el primer *byte* de B (b'\x08') es mayor al primer *byte* de C (b'\x03') por lo que intercambiamos todos los 3 por 5, y viceversa, para finalmente retornar $A\ B\ C\ 0$:

A = b'\x03\x02\x03'
B = b'\x08\x04\t'
C = b'\x05\x05'

Y el resultado final de la encriptación es la siguiente secuencia:

b'\x03\x02\x03\x08\x04\t\x05\x05\x00'

3.1.5. Ejemplo de codificación

En la figura 3 se muestra una representación de la estructura que debes enviar. En este caso se muestra cómo se manejaría un mensaje cualquiera donde el contenido son 210 *bytes*.

Supongamos que al codificar el contenido encriptado del mensaje a *bytes*, el resultado es una cadena de 210 *bytes*. Siguiendo el protocolo especificado, lo primero que se deberá enviar es un bloque de 4 *bytes* en *little endian* que indica el largo del contenido que se está enviando (210 *bytes*).

A continuación, se separan los 210 *bytes* del contenido encriptado en bloques de 80 *bytes*, resultando en 3 bloques en este ejemplo: los primeros dos bloques se envían de forma completa (80 x 2 = 160), pero como el tercero solo necesita enviar los 50 *bytes* restantes, se debe rellenar con 30 *bytes* 0 (`b'\x 00'`) para que el bloque alcance el largo pedido (80 *bytes*).

Después de esto, y para cada uno de los bloques resultantes, se deberá enviar primero el número del bloque correspondiente en un mensaje de 4 *bytes* en formato *big endian*, comenzando a contar desde el **0**. Posterior a este bloque, se debe enviar el bloque de 80 *bytes* que contiene parte del contenido. Una vez se hayan enviado todos los bloques, el proceso de envío habrá finalizado correctamente.

En la figura adjunta se muestra de forma gráfica la composición del *bytearray* que se debe obtener como resultado a partir de este proceso de codificación:

[2] Ojo: puede ser útil notar que las partes B y C no siempre tienen el mismo largo que la parte A.
[3] Recuerda que b'\x09' es equivalente a b'\t'.

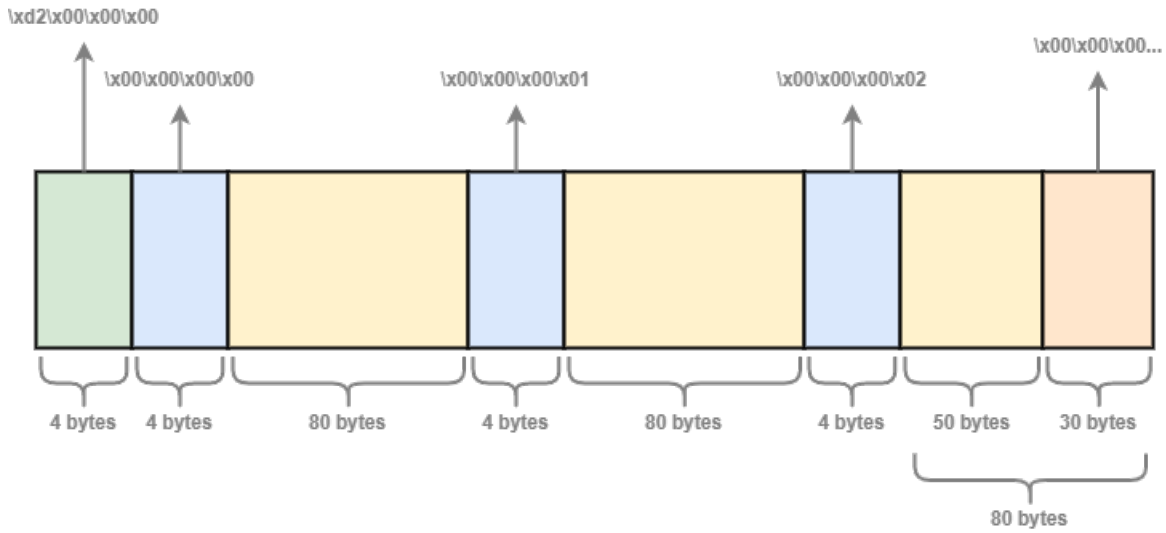


Figura 3: Ejemplo estructura del *bytearray* para un mensaje codificado.

3.1.6. *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:

- Se conecta un cliente al servidor. Debes indicar un identificador para este.
- Cuando un jugador reta a otro. Debes indicar un identificador para ambos jugadores.
- Cuando un jugador acepta un reto y comienza su partida. Debes indicar un identificador para ambos jugadores.
- 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 y número de turno.
- Un jugador confirma su apuesta. Debes indicar el nombre del jugador y la cantidad apostada.
- Se juega una ronda luego de que ambos jugadores confirmaron su apuesta, debes indicar el nombre de ambos jugadores y cual fue el ganador de la ronda.
- 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
-----------------|------------------------|------------------------
     Maxy15      |       Conectarse       |          -
-----------------|------------------------|------------------------
    EmiliaPinto  |     Aceptó el reto     |          -
-------------------------------------------------------------------
        -        |   Término de partida   | Ganador: Matiasmasjuan
-------------------------------------------------------------------
```

3.1.7. 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 la *Sala Principal*, entonces deja de ser considerado en la lista de jugadores conectados y el cupo queda disponible. Si se desconecta mientras está en un juego, entonces la partida termina automáticamente dejando como ganador al otro jugador.

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 dos jugadores comienzan una partida, el servidor deberá notificar de esto a todos los clientes conectados y se deberá reflejar en la ventana **Sala Principal** 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 cantidad de canicas que tiene cada uno de los clientes conectados y las actualizará una vez que se hagan las apuestas.

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 *DCCalamar*

4.1. Preparación

Antes de comenzar el juego todos los jugadores se encontrarán en la **Sala Principal**, en donde aparecerán los nombres de todos los jugadores junto con un botón para **retar** a ese jugador. Ten en cuenta que solo los nombres de los demás jugadores contarán con ese botón, es decir, no te podrás retar a ti mismo. En el momento de que alguien rete a algún jugador, tanto él como el jugador que reto pasarán a un **estado de espera** en el que no podrán ser retados ni retar a otros jugadores. El método para evitar que sean retados o que puedan retar a otros jugadores queda a criterio del estudiante, pero algunas opciones son:

bloquear los botones de reto, mostrar un mensaje en pantalla o utilizando un *pop up* que indique que no se puede retar a este jugador.

Al momento de retar a otro usuario, el jugador retado recibirá una notificación mediante una ventana o *pop up* donde podrá aceptar o rechazar la solicitud. En caso de aceptar la solicitud, ambos deberán dirigirse a la **Sala de juego** y la **Sala de espera** deberá actualizarse quitando a ambos competidores, de modo tal que otros usuarios no puedan enviar solicitudes a jugadores que se encuentran en una partida. Por el contrario, en caso de rechazar la solicitud, entonces tanto el jugador que retó como el retado dejarán de estar en **estado de espera** y podrán enviar solicitudes a otros usuarios.

4.2. Juego

DCCalamar consiste en un juego de canicas donde el objetivo será conseguir todas las canicas del oponente. Cada jugador comenzará con 10 canicas. El jugador que logre alcanzar 20 canicas dejando al oponente con 0 será el ganador.

El juego se basa en un sistema de turnos. Cada jugador deberá apostar una cierta cantidad de canicas y el usuario que juegue en su turno deberá adivinar si el oponente apostó una cantidad par o impar de canicas. En caso de acertar, deberá llevarse la misma cantidad de canicas que apostó el oponente. Por el contrario, si no acierta, el oponente se llevará las canicas que el jugador del turno apostó. Luego, el turno pasará al oponente, aplicándose el mismo procedimiento.

Al iniciar la partida, el jugador que comenzará adivinando será seleccionado de forma aleatoria ~~ya que todos somos iguales para el juego~~. A continuación, se repartirán 10 canicas a cada uno de los jugadores, y se dará por comienzo el duelo.

Es necesario destacar que la interfaz del juego de cada jugador debe ser distinta. Ambos jugadores deben de tener la opción de elegir cuántas canicas apostar en la ronda, el cual deberá estar dentro del intervalo entre 1 y el total de canicas que posea el jugador. Sin embargo, el jugador que se encuentre en su turno además deberá elegir si la apuesta del oponente es par o impar. Una vez ambos jugadores terminen de elegir sus opciones deberán marcar la opción de estar listos y se revelarán los resultados obtenidos, aplicándose las sumas y restas de canicas.

Luego se pasará a la siguiente ronda y se repetirá el mismo proceso con los roles invertidos, vale decir, ahora el segundo jugador será quien deberá adivinar si la apuesta del oponente es par o impar. Este mismo procedimiento continuará hasta que uno de los jugadores consiga las 20 canicas del oponente.

4.3. Fin del Juego

Una vez finalizado el juego se debe desplegar la Ventana final que mostrará quien es el ganador y quién el perdedor del juego. Además, debe tener la opción de jugar otra vez, el cual trasladará al jugador devuelta a la **Ventana de inicio**, donde deberá volver a registrarse para ingresar.

5. Interfaz gráfica

5.1. Ventana de inicio

Se despliega al iniciar el programa. Debe permitir que se introduzca el nombre del jugador, que será con el que se identificará en el servidor y en el juego con el resto de los jugadores. Con esto, los jugadores firman el contrato ~~y aceptan que están arriesgando su vida~~. La fecha de nacimiento y el nombre de usuario deben cumplir con las mismas condiciones señaladas en Flujo del programa. Si el nombre y la fecha de nacimiento son válidos, se prosigue con el flujo del juego a la sala principal, mientras que en caso contrario se debe notificar al usuario del error.

Figura 4: Ventana de inicio

5.2. Sala principal

Aparece luego de introducir el nombre de usuario y la fecha de nacimiento en la ventana de inicio. En la sala se despliegan todos los jugadores que se van uniendo a la partida actual. Para comenzar el juego, el usuario debe invitar a otro jugador que se haya conectado en el servidor a través del botón **RETAR**. Una vez hecho esto, el botón invitar para el jugador invitado debe ser deshabilitado para **el resto de los otros jugadores**, hasta que este conteste. Cuando el jugador invitado haya respondido afirmativamente, ambos jugadores son enviados a la sala de juego. El máximo de clientes que pueden estar conectados simultáneamente en la sala principal, para ser invitados por otros, es **igual a 4**. Sin embargo, solo **dos jugadores** pueden entrar a la misma sala de juego.

Figura 5: Ventana de la sala principal

5.3. Ventana de reto

Aparece cuando un jugador invita a otro en la sala principal. Debe aparecer el jugador que realizó la invitación y los botones **ACEPTAR** Y **RECHAZAR**. Al apretar cualquiera de las dos opciones, la ventana debe desaparecer. Si el jugador invitado aprieta la opción rechazar, se debe volver a habilitar el botón invitar destinado para él en la sala principal. Por otro lado, si aprieta el botón aceptar, debe redirigir a ambos a la sala de juego y también, ambos deben desaparecer de la sala principal.

Figura 6: Ventana de invitación

5.4. Sala de juego

Es la sala donde se desarrollará la partida, se debe tener, como mínimo, los siguientes elementos:

- Un área correspondiente a cada jugador, donde se muestre el avatar y el número de canicas que le quedan a cada uno. Este avatar debe ser designado aleatoriamente a cada jugador, procurando que sean distintos. La forma de mostrar la canicas quedará a criterio del estudiante: podrá ser mediante un número o con imágenes, tal como aparece en el ejemplo.
- Para el usuario que está jugando, debe aparecer una opción para indicar la cantidad de canicas a apostar en la ronda (en la figura, abajo de *¿Cuántas canicas apuestas?*) y si la apuesta hacia la cantidad de canicas que apostará el oponente es par o impar (en la figura, abajo de *La apuesta de tu oponente*). Además, una vez el usuario haya realizado sus apuestas, debe indicar que está **LISTO** para revelar la apuesta del oponente, una vez este también haya terminado su apuesta. Cabe destacar que este es **solo un ejemplo** de implementación. Quedará a criterio del estudiante si prefieren utilizar otro mecanismo, ya sea con un *Line edit* o un *Combo box*, siempre y cuando cumpla con el correcto funcionamiento.
- Para el jugador oponente, debe tener una sección donde aparezcan sus apuestas (en la figura, *¿Cuál es su apuesta* y *El valor de tu apuesta es*) y el resultado de la ronda una vez revelada la información (en la figura, cuadrado gris). Una vez terminada la ronda, se debe actualizar la cantidad de canicas de cada jugador.
- El área donde aparece el resultado de la ronda también debe indicar cuando un jugador no ha realizado su apuesta.

Figura 7: Ventana de la sala de juego (antes de que el oponente juegue)

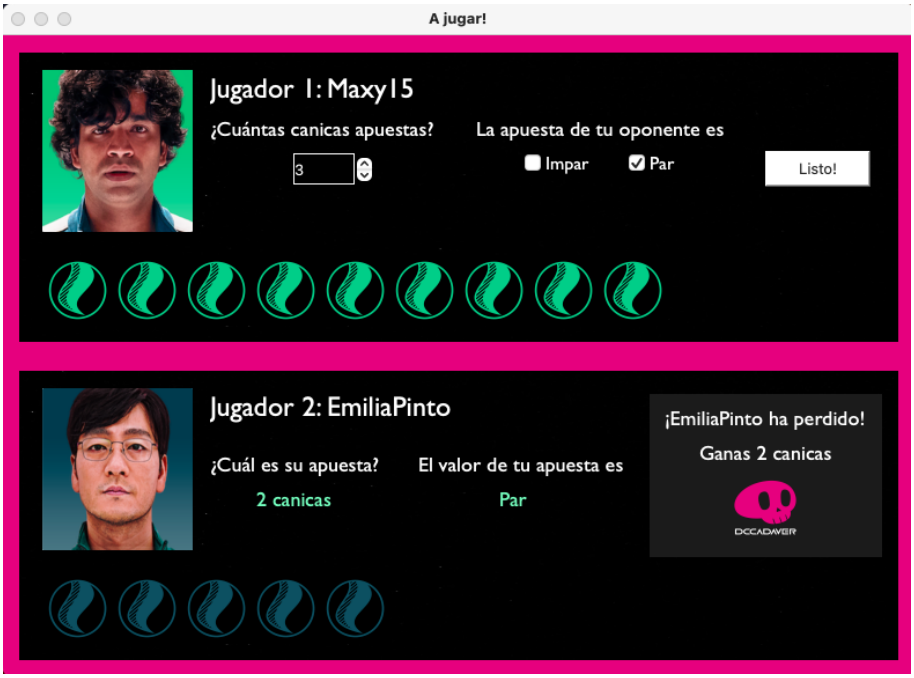


Figura 8: Ventana de la sala de juego (luego de que el oponente juegue)

5.5. Ventana final

Esta ventana muestra al jugador ganador y al perdedor, indicando cuál es el estatus de cada uno, es decir, si sobrevivió o no al *DCCalamar*. Además, posee el botón **JUGAR OTRA VEZ** que redirigirá a la ventana de inicio, así los jugadores pueden volver a conectarse a la sala principal e iniciar otra partida. Importante destacar que no se puede volver a la sala principal desde la ventana final, por lo que será necesario volver a conectarse desde la ventana de inicio.

Figura 9: Ventana final

6. Archivos

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

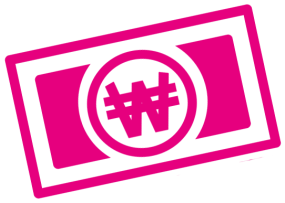
6.1. Archivos entregados

6.1.1. Sprites

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

- **Avatares:** Contiene los distintos avatares que deberás asignar a cada jugador de la partida, su formato es de la forma avatar-x, siendo x el número del jugador correspondiente.
- **Decoraciones:** Contiene *sprites* para que decores tus ventanas, como calaveras con distintos colores, el billete en wones y las figuras de la tarjeta de invitación.
- **Juego:** Contiene las canicas pertenecientes a cada jugador, diferenciadas por color y que es el mismo que el fondo del avatar respectivo. Su formato es canicas-x, siendo x el número del jugador correspondiente.
- **Logos:** Contiene el logo del juego en diferentes colores y fondos, para que uses el que más te guste.

(a) *Sprite* para avatar de juego



(b) *Sprite* para decorador

Figura 10: 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**[4].
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 (3 décimas)

Si notas que el juego no va a tu favor, puedes decidir estafar a tu compañero para ser el próximo ganador. Para esto, decides implementar **2 cheatcoddes**:

- **F + A**: Al presionar esta combinación de teclas, el jugador que se encuentra en el turno podrá convencer a su compañero para que le diga la cantidad de canicas que apostará, de tal manera que sabrás el número exacto y podrás seleccionar la paridad correcta. Esta opción sólo debe funcionar después de que el oponente haya presionado el botón de *Listo*. Quedará a tu criterio la forma de mostrar el número de canicas del oponente, ya sea con un mensaje en la ventana o con un *popup*.

[4]Esta nota es sin considerar posibles descuentos.

- **Z + X**: Al presionar esta combinación de teclas, el jugador que se encuentra en el turno podrá robarle 3 canicas a su compañero automáticamente, lo cual puede o no implicar su victoria inmediata.

7.2. Turnos con Tiempo (2 décimas)

Como ~~en la serie~~ solo tienen 30 minutos para decidir un ganador, decides que lo mas lógico es poner un temporizador por turno para que ningún jugador se tome demasiado tiempo en elegir su apuesta.

Para tener el puntaje de este bonus debes implementar un timer de `TIEMPO_TURNO` segundos que se inicie automáticamente al comenzar una ronda y que verifique que los jugadores elijan sus apuestas antes de que se termine el tiempo (se consideran elegidas las apuestas cuando el jugador aprieta el botón de listo). si un jugador se le acaba el tiempo, ~~será brutalmente ejecutado~~ perderá automáticamente la partida.

8. Avance de tarea

Para esta tarea, el avance corresponde a implementar el **inicio de la interacción con** *DCCalamar*, 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 Sala principal, de modo tal que el usuario pueda interactuar con el servidor y tener acceso a la Sala Principal desde la Ventana de Inicio siempre y cuando cumpla con las condiciones anteriores.

Para fines de este avance, no es necesario implementar las solicitudes de enfrentamiento en la Sala Principal, ni una actualización en vivo de los usuarios dentro de ella. Tampoco es necesario que la información que sea enviada entre cliente y servidor siga el protocolo de envío. Quedará a tu criterio si deseas 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).