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:

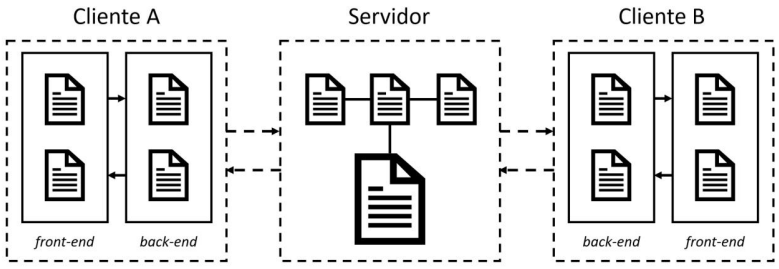

Figura 2: Separación cliente-servidor y *front-end-back-end*.

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'.

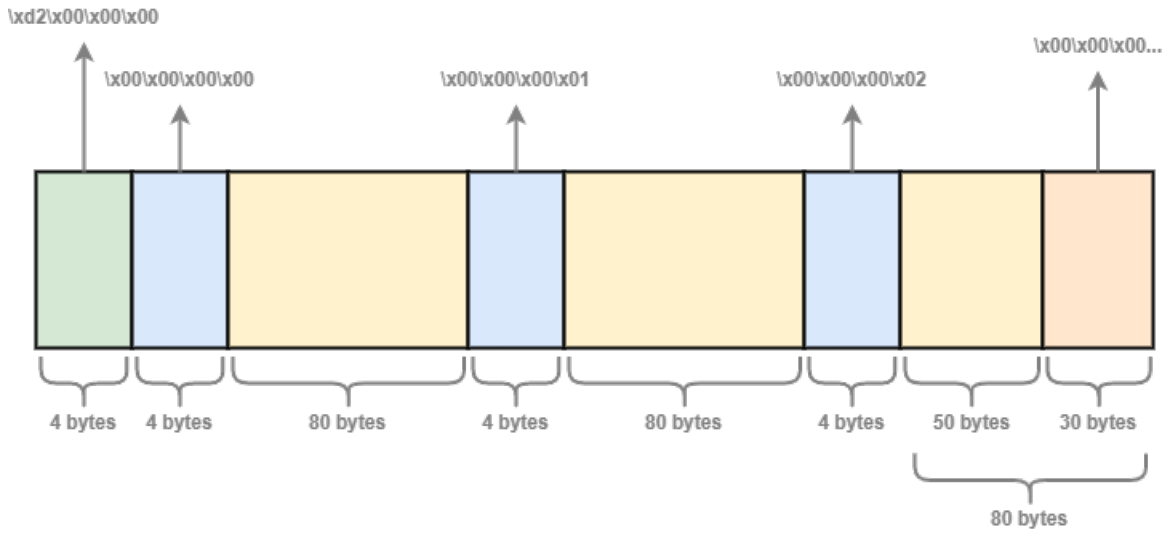

Figura 3: Ejemplo estructura del *bytearray* para un mensaje codificado.

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: