



PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE  
ESCUELA DE INGENIERÍA  
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2233 Programación Avanzada (2023-1)

## Tarea 3

### Entrega

- Tarea y README.md
  - Fecha y hora: martes 20 de junio de 2023, 20:00
  - 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

<b>1.</b> <i>DCCachos</i>	<b>3</b>
<b>2.</b> Flujo del programa	<b>3</b>
<b>3.</b> <i>Networking</i>	<b>4</b>
3.1. Arquitectura cliente - servidor . . . . .	4
3.1.1. Separación funcional . . . . .	4
3.1.2. Conexión . . . . .	5
3.1.3. Método de codificación . . . . .	5
3.1.4. Método de encriptación . . . . .	6
3.1.5. Ejemplo de envío de mensaje . . . . .	7
3.1.6. <i>Logs</i> del servidor . . . . .	8
3.1.7. Desconexión repentina . . . . .	8
3.2. Roles . . . . .	9
3.2.1. Servidor . . . . .	9
3.2.2. Cliente . . . . .	9
<b>4.</b> Reglas de <i>DCCachos</i>	<b>9</b>
4.1. Preparación del juego . . . . .	9
4.2. Inicio de juego . . . . .	9
4.3. Valor de dados . . . . .	10
4.4. Turno . . . . .	10
4.4.1. Anunciar valor . . . . .	10
4.4.2. Pasar turno . . . . .	10
4.4.3. Dudar . . . . .	10
4.4.4. Cambiar dados . . . . .	11
4.4.5. Usar poder . . . . .	11
4.5. <i>Bots</i> . . . . .	11
<b>5.</b> Interfaz gráfica	<b>11</b>
5.1. Ventana de inicio . . . . .	12
5.2. Ventana de juego . . . . .	13
<b>6.</b> Archivos	<b>14</b>
6.1. Archivos entregados . . . . .	14
6.1.1. <i>Sprites</i> . . . . .	14
6.1.2. <i>Scripts</i> . . . . .	14
6.2. Archivos a crear . . . . .	14
6.2.1. <i>parametros.json</i> . . . . .	14
6.2.2. <i>main.py</i> . . . . .	15
<b>7.</b> <i>Bonus</i>	<b>15</b>
7.1. SEE (1 décimas) . . . . .	15
7.2. Turno con tiempo (3 décimas) . . . . .	15
<b>8.</b> <i>.gitignore</i>	<b>15</b>
<b>9.</b> Entregas atrasadas	<b>16</b>
<b>10.</b> Importante: Corrección de la tarea	<b>16</b>
<b>11.</b> Restricciones y alcances	<b>17</b>

## 1. *DCCachos*

Luego de haber rescatado exitosamente al mago **Aaossa** ante los fantasmas de la mansión embrujada del malvado **Dr. Pinto**, decides emprender el viaje de regreso al castillo del gran **Rey Joaking**. Al momento de llegar, te das cuenta que tanto el gran **Rey Joaking**, la gobernadora **Lily416** y el caballero **Sir Hernan4444** se encuentran reunidos para elaborar una estrategia y derrotar al malvado **Dr. Pinto** de una vez por todas.

Ante esto, el mago **Aaossa** te informa que durante su encierro descubrió que el malvado **Dr. Pinto** tiene cierta adicción a las apuestas, y se las pasaba jugando a los *DCCachos* junto a sus amigos fantasmas. Al escuchar esta noticia, se te ocurre la brillante idea de vencer al malvado **Dr. Pinto** en su propio juego, pero para esto, necesitarás la ayuda de tus aliados: **Lily416**, **Sir Hernan4444** y el gran **Rey Joaking**.

Es por esta razón que decides como estudiante de *Programación avanzada*, utilizar tus conocimientos sobre *serialización* y *networking*, para realizar un programa que les permita luchar en conjunto en el gran juego *DCCachos*, y de esta forma acabar finalmente con el malvado **Dr. Pinto**.



Figura 1: Logo de *DCCachos*

## 2. Flujo del programa

*DCCachos* es un juego multijugador por turnos que consiste en engañar a tus rivales usando las probabilidades hasta ser el último jugador con vida. Para ello, cada jugador tendrá una cantidad de **NUMERO\_VIDAS** vidas iniciales, las cuales podrán ir variando durante el juego. Al inicio de cada ronda los jugadores deberán lanzar dos dados, los cuales se mantendrán secretos ante los demás. Durante el turno, los jugadores podrán especular sobre las declaraciones del jugador anterior, ya sea subiendo el valor actual o dudando de la jugada anterior. Cualquiera de estas decisiones podrá repercutir en las vidas del jugador, en donde el último que quede con vidas 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 ejecutar tu programa, el jugador deberá visualizar la **Ventana de inicio**, en el cual se le asignará de forma automática un nombre de usuario dentro de los disponibles en **parametros.json**. Además, en la ventana se deberá desplegar los nombres de todos los usuarios ya conectados. Si alguno de los jugadores presiona el botón para **comenzar** la partida, entonces deberás dirigirte a la **Ventana de juego**. Si este botón se presiona, pero aún no se encuentran **NUMERO\_JUGADORES**<sup>1</sup> jugadores conectados, entonces se deberán llenar los jugadores restantes con **Bots**.

<sup>1</sup>Este valor será equivalente a 4 en todo momento. Sin embargo, deberá ser importado y utilizado a través de **parametros.json**

En la [Ventana de juego](#) se mostrarán los nombres de los jugadores, los números de turno y las apuestas. Además, podrás interactuar con la ventana para lanzar tus dados, aumentar una apuesta o dudar. El juego estará dado por turnos y el orden inicial será dado aleatoriamente. El resto de las reglas serán indicadas en la sección [Reglas de \*DCCachos\*](#). Una vez que solo quede un jugador con vida, se deberá desplegar en la Ventana de juego un texto mostrando si ganaste o perdiste el juego.

### 3. *Networking*

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

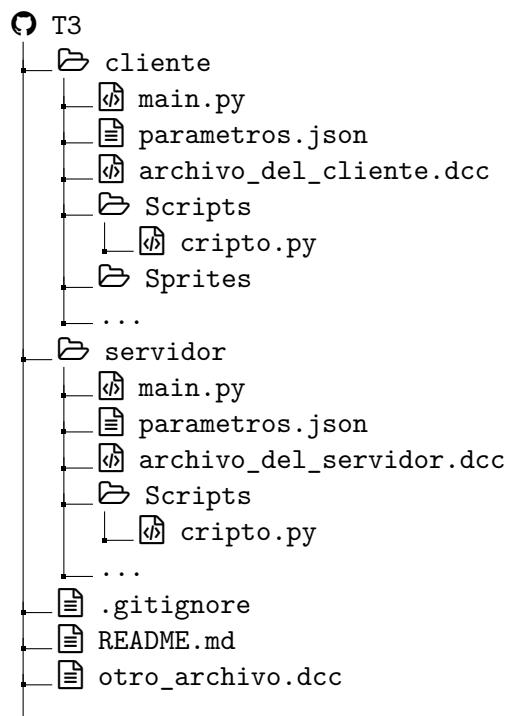
Tu misión será implementar dos programas separados, uno para el **servidor** y otro para el **cliente**. De los mencionados, **siempre** se deberá ejecutar primero el servidor y este quedará escuchando para que se puedan conectar uno o más clientes. Debes tener en consideración que la comunicación es 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. Si algo no está especificado o si no queda completamente claro, puedes [preguntar aquí](#).

##### 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:



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:

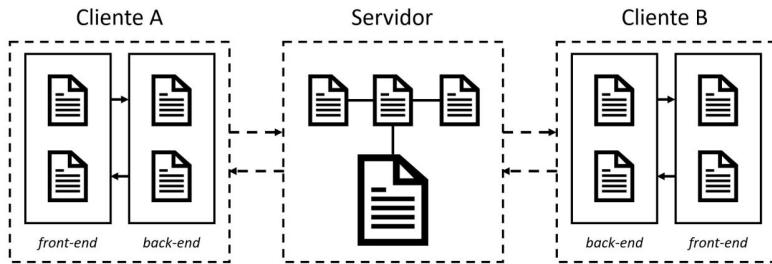


Figura 2: Separación cliente-servidor y *front-end-back-end*.

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 carácter **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 el host para instanciar un *socket*. El archivo debe llevar el siguiente formato:

```

1  {
2      "host": <dirección_ip>,
3      ...
4  }
```

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**. La selección del puerto, tanto para el cliente como para el servidor, del *socket* se puede encontrar en [main.py](#). Es importante recalcar que el **cliente** y el **servidor no deben usar el mismo archivo JSON** para obtener los 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 el valor del dado que es lanzado y este le responderá con un mensaje indicando los efectos que tuvo este resultado en el juego. Sin embargo debemos tener cuidado, ya que si un contrincante intercepta un mensaje y lo lee, sabrá qué números tenemos.

Para evitar lo anterior, debemos asegurarnos de **encriptar** el contenido del mensaje 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* indican el **largo del mensaje encriptado**, los cuales deben ser enviados en formato **little endian**<sup>2</sup>.
- Después, se debe separar el mensaje encriptado en *chunks* de 128 *bytes*, los cuales deben ser precedidos por otros 4 *bytes* que indiquen el número de bloque enviado partiendo desde el cero y codificados en **big endian**.
- Si no se logra llenar un *chunk* con 128 *bytes* deberás llenar con bytes ceros (`b'\x00'`).

Finalmente, el mensaje completo que se envía se verá de la siguiente forma:

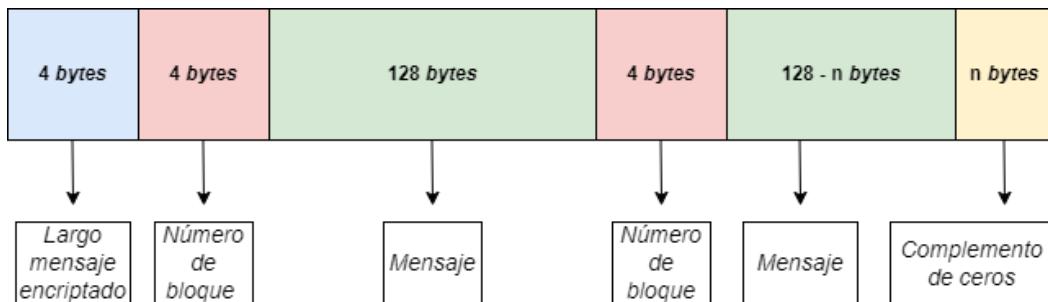


Figura 3: Ejemplo de un *bytearray* codificado.

### 3.1.4. Método de encriptación

Finalmente, el **método de encriptación** que implementarás para detener a los trampagos y derrotar al malvado **Dr. Pinto** es el siguiente:

- Dentro del archivo `parametros.json`, deberá definirse un valor llamado **N\_PONDERADOR**, el cual corresponde a un valor **int** entre **1** y **100**, que se mantendrá **constante** en la ejecución del programa.
- Luego, se deben mover los *bytes* del mensaje una cantidad de **N\_PONDERADOR** espacios a la derecha. Por lo tanto, cada *byte* en una posición Y del mensaje original, pasará a estar en la posición  $(Y + N\_PONDERADOR)$  del mensaje encriptado. Los **N\_PONDERADOR** *bytes* finales como no pueden ser movidos a la derecha, se moverán al inicio del *bytearray*.

```
bytearray_original[Y] = bytearray_encriptado[(Y + N) mod len(bytearray_original)]
```
- Finalmente, se debe intercambiar el valor del `bytearray_encriptado[N]` con `bytearray_encriptado[0]`, es decir, el primer carácter del mensaje va a ser el carácter **N\_PONDERADOR** y viceversa.

<sup>2</sup>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 quieras usar para representarlo. Para más información puedes revisar este [enlace](#).

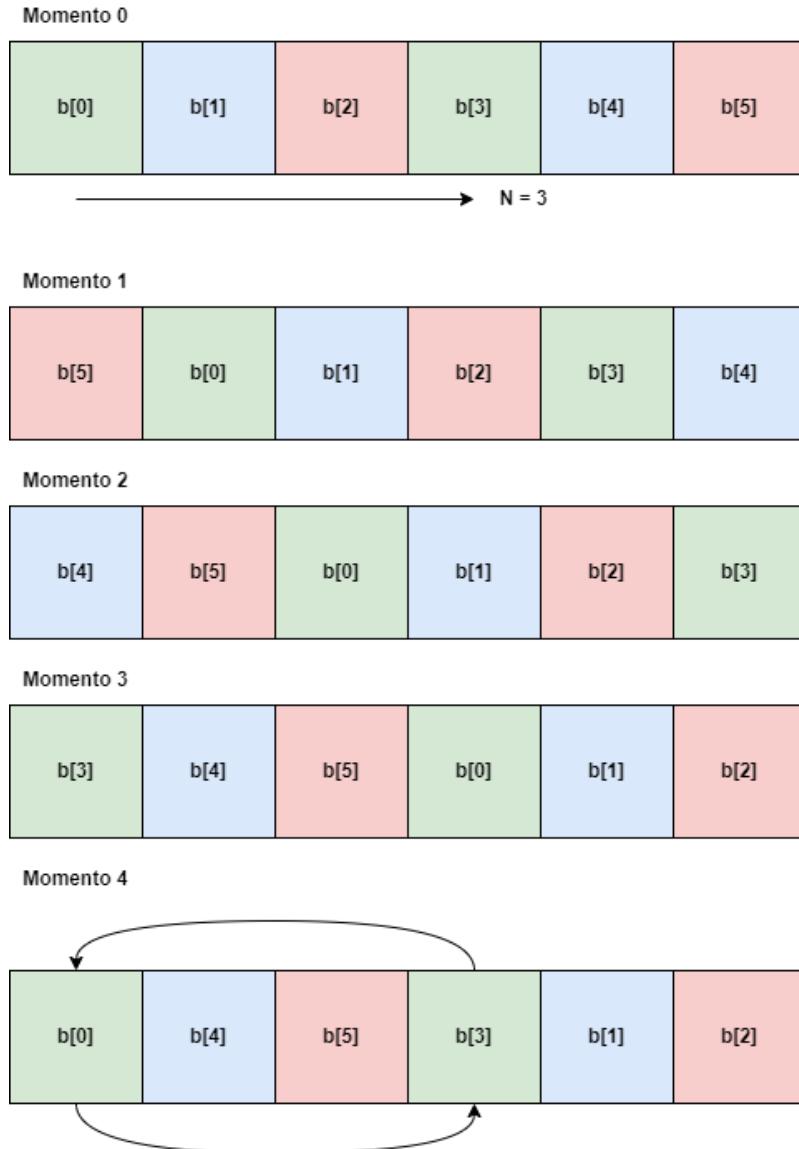


Figura 4: Pasos del proceso de encriptación.

### 3.1.5. Ejemplo de envío de mensaje

Supongamos que desde el cliente le queremos un mensaje al servidor que sea el *string* '**Hola mundo**'. Los pasos que deberíamos tomar son los siguientes:

1. Serializar el mensaje con `json.dumps` o `pickles.dumps`.
2. Tomando la serialización de `pickle` como ejemplo, resultaría el siguiente:  

```
\x80\x04\x95\x0e\x00\x00\x00\x00\x00\x00\x00\x8c\nHola mundo\x94.
```
3. Supongamos que tenemos un parámetro `N_PONDERADOR` igual a **17**. Por lo tanto, debemos mover los *bytes* **17** veces a la derecha.
4. Todo *bytes* es desplazado 17 veces a la derecha, y aquellos que su posición que superen el largo del mensaje, se deben incorporar al inicio del mensaje:  
(Movimiento 1) `\x94.\x80\x04\x95\x0e\x00\x00\x00\x00\x00\x00\x8c\nHola mundo`

(Movimiento 2) o\x94.\x80\x04\x95\x0e\x00\x00\x00\x00\x00\x00\x8c\nHola mund

(Movimiento 3) do\x94.\x80\x04\x95\x0e\x00\x00\x00\x00\x00\x00\x8c\nHola mun

Y esto sigue hasta lograr los 17 movimientos:

(Movimiento 17) \x00\x00\x00\x8c\nHola mundo\x94.\x80\x04\x95\x0e\x00\x00\x00'

5. Finalmente, el *byte* de la posición 17, se debe intercambiar con el *byte* que quedó en la posición 0.

\x80\x00\x00\x8c\nHola mundo\x94.\x00\x04\x95\x0e\x00\x00\x00'

6. Ahora sólo falta codificarlo: primero hacemos un *chunk* de 4 *bytes* que indique el largo de 25 bytes, seguido de otro que diga que es el *chunk* 1.

7. Como el mensaje sólo tiene 25 *bytes* hay que llenar con 103 *bytes* cero a la derecha.

8. Ya con todos los *chunks* listos, los concatenamos y los enviamos al servidor.

Finalmente, para decodificar y obtener el mensaje original, el receptor deberá realizar los pasos inversos a los especificados.

### 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.
- Cuando se comienza una partida. Debe indicar quienes están en esa partida.
- Comienza el turno de un cliente. Debes indicar su nombre.
- Cuando un jugador realiza una jugada dentro de las disponibles en `Turno`.
- Cuando un jugador pierde una vida. Se debe escribir las que le quedan.
- Se termina la partida, debes indicar quién 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
MIrarraZaval	Conectarse	-
matiasmasjuan	Pasa su turno	-
-	Término de partida	Ganador: CrisDonosoMaass

### 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 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 un jugador 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 valor de sus datos 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 termina su turno y continua el siguiente, el servidor deberá notificar de esto a todos los clientes en la partida y se reflejará 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 las vidas y turnos de cada uno de los clientes conectados y las actualizará una vez que se termine el turno o ronda.

### 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 *DCCachos*

### 4.1. Preparación del juego

Antes de empezar una partida, todos los jugadores se encontrarán en la [Ventana de inicio](#), en la cual se reúnen y podrán ir conectándose y desconectándose respetando el límite de [NUMERO\\_JUGADORES](#). Una vez que algún jugador decida empezar la partida, entonces los jugadores presentes se redirigirán a la [Ventana de juego](#). Si algún jugador decide empezar con un número menor a [NUMERO\\_JUGADORES](#) presentes, entonces los jugadores faltantes serán **reemplazados** por [Bots](#), cuyo comportamiento se explicará más adelante.

Si un jugador intenta conectarse a una partida que ya está en juego o donde ya están conectados [NUMERO\\_JUGADORES](#) jugadores, este jugador **no** se conectará a la partida. En cambio, el jugador se mantendrá en la ventana de inicio y recibirá un *pop-up* que le informe que espere a la próxima partida.

### 4.2. Inicio de juego

Para jugar *DCCachos*, los jugadores se encontrarán posicionados formando un círculo, donde cada uno tendrá dos dados y un marcador con su cantidad de vidas restantes. El juego se separa por rondas que

solo terminan cuando algún jugador haya perdido una vida. Si un jugador se queda sin vidas, este se verá eliminado del juego. El objetivo del juego es ser el último jugador en pie.

Al inicio de la partida, cada jugador parte con **NUMERO\_VIDAS** vidas iniciales y se elige un jugador al azar para que empiece la partida. Los jugadores toman turnos en sentido antihorario partiendo por este jugador. Cuando un jugador pierda una ronda, éste será el que tendrá el primer turno en la siguiente ronda. En el caso de que este jugador haya sido eliminado, empieza el siguiente jugador al que perdió según sentido antihorario.

Al principio de cada ronda, todos los jugadores lanzarán sus dados. Este proceso será realizado automáticamente por el **servidor**, por lo que al comienzo de la ronda, los jugadores partirán con un nuevo número en los dados. Estos deberán ser vistos **únicamente** por el mismo jugador, vale decir, no podrás ver los dados de tus otros oponentes.

### 4.3. Valor de dados

Cada *set* de dos dados tiene un valor asociado. Este valor se calcula como la suma de ambos, siendo los valores más altos los más “fuertes”. Al principio de la ronda, el jugador que comienza debe indicar algún valor inicial, y cada jugador en su turno debe anunciar un valor **estrictamente** más **grande** al jugador anterior. Existen ciertas combinaciones de dados que le dan habilidades adicionales al jugador al que pertenecen. El valor de estas manos seguirá siendo la suma de sus dados a pesar de su carácter especial.

### 4.4. Turno

Durante el turno de un jugador, podrá realizar las siguientes opciones:

1. Anunciar un valor más alto.
2. Pasar turno.
3. Dudar del jugador anterior.
4. Cambiar dados.
5. Usar poder.

#### 4.4.1. Anunciar valor

Al anunciar un valor, un jugador debe decir un valor de dados a todos los otros jugadores, sin mostrar que dados realmente tiene. Este valor debe ser **estrictamente** más **grande** que el del jugador anterior y no necesariamente el **mismo** valor que tiene en sus dados. Si el jugador anuncia un valor más alto del que en verdad tiene, se considera que este está “mintiendo”. Mentir no significa perder una vida o hacer trampa, pero se arriesga a que el próximo jugador dude de su turno.

#### 4.4.2. Pasar turno

Si un jugador anuncia que va a pasar, el turno avanza hacia el próximo jugador, **conservando** el valor declarado por el jugador anterior al que pasó. Un “paso” solo se considera válido si el jugador que lo anuncia tiene dos dados que suman **exactamente** **VALOR\_PASO**. En otro caso, también se considera que ese jugador estaba “mintiendo”.

#### 4.4.3. Dudar

Si un jugador sospecha que el jugador anterior estaba mintiendo, este tiene la oportunidad de **dudar** su turno. Después de dudar, todos los jugadores revelan sus dados de tal forma que cualquier jugador pueda

ver los dados de los demás. Si el jugador anterior estaba efectivamente mintiendo, este pierde una vida. En caso contrario, el jugador que dudó es el que pierde la vida. Una vez disminuida la vida del jugador que perdió, todo los jugadores tiran sus dados de nuevo y empieza una nueva ronda.

Solamente se puede dudar la acción realizada **inmediatamente** antes del turno actual. Por ejemplo, si el jugador 1 da un valor de 5 y el jugador 2 pasa de turno, el jugador 3 **solamente** puede dudar que el valor del jugador 2 sea distinto de **VALOR\_PASO** y no puede dudar que el valor del jugador 1 sea menor a 5.

#### 4.4.4. Cambiar dados

El jugador tiene la oportunidad de tirar sus dados nuevamente para tratar de obtener un valor más alto. Esto solo se puede hacer como máximo **una vez por turno** y el jugador **no va a poder dudar** del jugador anterior luego de haber cambiado sus dados. Para terminar su turno, el jugador debe realizar alguna de las otras acciones disponibles utilizando sus nuevos dados, vale decir: **anunciar valor, pasar turno o usar poder**.

#### 4.4.5. Usar poder

Estos se podrán activar cuando el jugador tiene una combinación de dados específica. El jugador que busca usar un poder debe revelar sus dados a los otros jugadores y elegir a algún jugador (que puede ser él mismo) a quien aplicar el poder. No se puede mentir durante el uso de un poder y la ronda se acaba inmediatamente después de aplicarlo.

- **¡Ataque!**: Si un jugador tiene exactamente 1 y 2 (o 2 y 1), entonces este jugador puede levantar su vaso y anunciar “¡Ataque!”. En este caso, el jugador debe elegir a otro a quien disminuirle el contador de vida en 1.
- **¡Terremoto!**: Si un jugador tiene exactamente 1 y 3 (o 3 y 1), entonces este jugador puede levantar el vaso y anunciar “¡Terremoto!”. En este caso, el jugador debe elegir a otro a quien le cambia el contador de vida a un valor al azar entre 1 y **NUMERO\_VIDAS**, inclusive.

### 4.5. Bots

Los **Bots** son jugadores que serán simulados de forma automática y estarán controlados internamente por el servidor. A diferencia del jugador, un *bot* **nunca** va a ocupar un poder y siempre va a seguir el mismo patrón. Cuando sea el turno del *bot*, seguirá el siguiente procedimiento, en el mismo orden, para realizar la jugada.

1. El *bot* tiene una probabilidad **PROB\_DUDAR** de dudar el jugador anterior.
2. Si es que el *bot* no duda, entonces el *bot* inmediatamente cambia sus dados.
3. Una vez cambiados los dados, el *bot* trata de anunciar un número con probabilidad **PROB\_ANUNCIAR**. Si el *bot* decide anunciar un número, este anuncia un número al azar en el rango [*valor jugador anterior* + 1; 12]. Si el valor obtenido es igual a 12, entonces el *bot* no anuncia un valor.
4. Si es que el *bot* no anuncia un valor, entonces se ve obligado a pasar de turno.

## 5. Interfaz gráfica

A continuación se presentan las distintas ventanas que deberán implementar para este juego. El método de creación de estas ventanas queda a tu criterio, ya sea utilizando únicamente Python o apoyarse de

QtDesigner. En caso de utilizar esta última herramienta, no olvides subir el archivo .ui junto con tu código, o de lo contrario, tu evaluación **no podrá ser evaluada**.

### 5.1. Ventana de inicio

Esta es la ventana principal del juego y se ejecuta al iniciar el programa. Al comenzar, no se pide por pantalla un nombre de usuario, sino que se asigna aleatoriamente entre los nombres almacenados en `parametros.json` y estos **no deben repetirse**. Los jugadores que ingresan y salen de la ventana se deben actualizar en las ventanas de todos los clientes conectados.

Si un cliente se intenta conectar y la sala ya se encuentra llena o hay una partida en curso, se le debe avisar mediante una ventana o *pop-up* debido a la razón correspondiente. Si uno de los jugadores pulsa la opción de comenzar, entonces todos los jugadores se redirigirán a la [Ventana de juego](#). En caso de que no se completen todos los jugadores, los faltantes serán cubiertos por *Bots*.

Elementos mínimos de la Ventana de inicio son:

1. Título con un *label* de **Sala de Espera**.
2. Para los clientes conectados en la sala, debes mostrar al jugador con su respectivo nombre. No olvides considerar los casos cuando la sala está llena o con una partida en juego como se indica en [Ejemplo de Ventana de inicio y pop-ups..](#)
3. Opción para comenzar partida.
4. Opción para salir del programa.



Figura 5: Ejemplo de Ventana de inicio y *pop-ups*.

## 5.2. Ventana de juego

Esta ventana es la sala en donde se desarrollará la partida de *DCCachos*. Como la duración de una ronda está en función de quién pierda una vida, al inicio de cada ronda, todos los jugadores lanzaran sus dos dados automáticamente. Cada cliente tiene acceso a un menú de acciones mediante el uso de botones y un `QLineEdit`.

Para el caso del cliente que inicia la ronda, este puede anunciar el primer valor sin la restricción del valor estrictamente mayor al anterior, este valor se debe ingresar mediante un `QLineEdit` y su respectivo botón. Si posee `VALOR_PASO` puede anunciar aquel valor o pulsar botón de pasar turno. Si el jugador cree no tener buenos dados, puede lanzar nuevamente pulsando el botón de cambiar dados una vez por turno. Si obtiene los dados para utilizar un poder, el botón de usar poder se activara solo si posee los dados correspondientes enunciados en [Reglas de DCCachos](#). Al ser el primer jugador, no podrá dudar del anterior. Todos los cambios referente a los turnos y valores anunciados serán mostrados por la interfaz como se indica en [Ejemplo de Ventana de juego](#). en la parte superior de la ventana.

Para los siguientes turnos de la ronda se aplicará la restricción del valor estrictamente mayor al anunciado por el jugador anterior, en el caso de no cumplir se debe indicar el error por interfaz utilizando el `QLineEdit` implementado y solicitar ingresar un valor correcto. Se podrá dudar sin problemas del jugador anterior, cuando esto ocurra, todos los jugadores deberán mostrar sus dados y aplicar las reglas respectivas de quien pierde la vida. Cada vez que cambie algún valor, se deben evidenciar todos los cambios en los *labels* de la interfaz. Se puede utilizar los *sprites* de las caras de los dados para indicar la cantidad de vidas.

Si un jugador queda sin vidas, entonces éste deberá ser notificado por un `QLabel` o *pop-up* (el cual aparecerá al centro de la ventana) que indicará su derrota y que debe salir del programa. Los clientes que sigan conectados, seguirán jugando hasta terminar la partida. Para quien gane la partida, deberá ser notificado por un `QLabel` o *pop-up* (el cual aparecerá al centro de la ventana) indicando su victoria y que debe cerrar el programa.

En una partida normal no se muestran los dados de los demás, es decir, estos estarán ocultos a menos que se dude. Por lo que si aplicas el *bonus SEE (1 décimas)*, se deberán mostrar los dados con los respectivos *sprites* entregados. Para el bonus de *Turno con tiempo (3 décimas)*, puedes utilizar un `QLabel` al centro del círculo para indicar la cuenta regresiva.

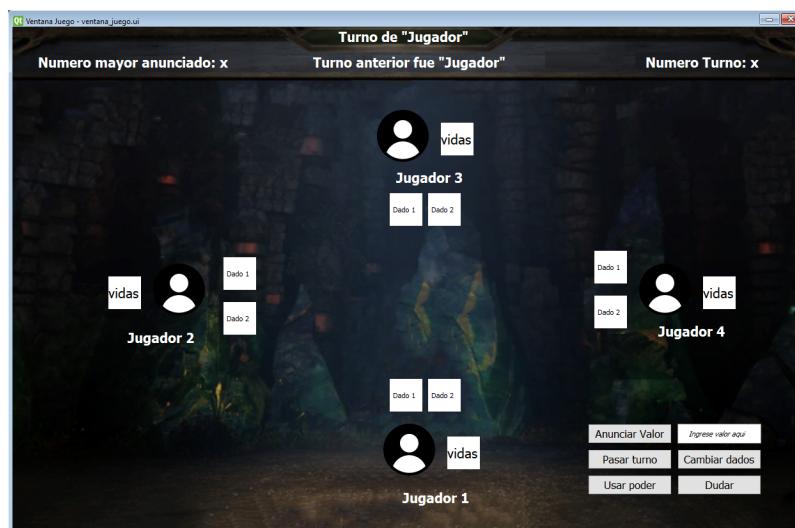


Figura 6: Ejemplo de Ventana de juego.

## 6. Archivos

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

### 6.1. Archivos entregados

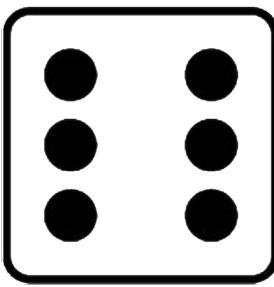
#### 6.1.1. *Sprites*

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

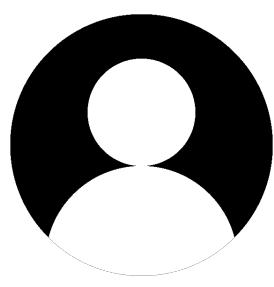
- **background**: Contiene los *sprites* para los fondos de las ventanas que se pide implementar.
- **dices**: Contiene los *sprites* de los dados.
- **extra**: Contiene algunos *sprites* extra que facilitar la lógica del diseño de las ventanas.



(a) Carpeta `background`.



(b) Carpeta `dices`.



(c) Carpeta `extra`.

Figura 7: Ejemplo de los *sprite* de cada carpeta.

#### 6.1.2. *Scripts*

Carpeta que contiene todos los *scripts* extras que necesitas para llevar a cabo tu tarea, entre ellos encontrarás:

- **cripto.py**: Contiene **dos funciones que deberás completar** para testear la correcta encriptación y desencriptación del mensaje respectivamente. Este archivo se utilizará para evaluar los ítems asociados a la correcta implementación del [Método de encriptación](#), además de ser utilizado en la comunicación **cliente - servidor**. Este archivo deberá estar presente tanto en la carpeta de servidor como del cliente<sup>3</sup>. Para su corrección, se ejecutará **únicamente el archivo presente en la carpeta del cliente**.

## 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* de los *sprites*; mientras que `parametros.json` de `servidor` contendrá solamente parámetros útiles para el servidor.

El archivo `parametros.json` del servidor deberá contener nombres por defecto para que todos los usuarios de una partida tengan un nombre asignado. A diferencia de las tareas pasadas, los parámetros en

---

<sup>3</sup>Deberás duplicar este archivo.

**ESTE FORMATO** deberán ir en este archivo, junto con su respectivo valor. Los archivos deben encontrarse en formato JSON (**NO un archivo de extensión .py**), como se ilustra a continuación:

```
1 {
2     "host": <direccion_ip>,
3     "id_1": "Lily416",
4     "id_2": "Sir Hernan4444",
5     "id_3": "Rey Joaking",
6     "id_4": "Dr. Pinto",
7     ...
8 }
```

### 6.2.2. main.py

Dentro de la tarea debe existir un archivo central para su ejecución por consola, tanto para el cliente como para el servidor. La ejecución del programa deberá aceptar argumentos de línea de comando<sup>4</sup> para poder seleccionar el puerto de conexión del *socket*, como se ilustra a continuación:

```
1 python main.py <port>
```

## 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**<sup>5</sup>.
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 4 décimas. Deberás indicar en tu README si implementaste alguno de los *bonus*, y cuáles fueron implementados.

### 7.1. SEE (1 décimas)

Al presionar **consecutivamente** las teclas **S+E+E** se deberán mostrar los datos de todos los jugadores de la mesa. Esto solo aplica únicamente para el jugador que ingresó la clave.

### 7.2. Turno con tiempo (3 décimas)

Al comenzar cada turno, el jugador tendrá un tiempo de **TIEMPO\_TURNO** segundos para realizar su jugada. Si el jugador no termina su turno transcurrido este tiempo, entonces perderá una vida y comenzará una nueva ronda. Cabe recalcar que el tiempo debe ser visible e ir disminuyendo cada segundo que transcurre.

## 8. .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](#).

Los archivos a ignorar para esta tarea son:

---

<sup>4</sup>Se recomienda usar `sys.argv`.

<sup>5</sup>Esta nota es sin considerar posibles descuentos.

- Enunciado.pdf
- Carpeta de *Sprites* entregada junto al enunciado.

Recuerda **no ignorar tu archivo de parámetros, cripto y archivos .ui, o tu tarea no podrá ser revisada.**

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.

## 9. Entregas atrasadas

Entregas con atraso tendrán una penalización en la calificación máxima obtenida acorde a la política de atrasos. Sin embargo, tendrás a disposición cupones para eliminar días de atrasos.

**Política de atraso:**

- Hasta 1 día de atraso (00:01 a 24:00 hrs de atraso) tendrán calificación máxima 6,0.
- Hasta 2 días de atraso (24:01 a 48:00 hrs de atraso) tendrán calificación máxima 4,0.
- Despues de las 48 hrs, no se aceptarán entregas atrasadas y se aplicará calificación mínima 1,0.

**Cupones:**

- Durante el semestre cada estudiante dispondrá de 2 cupones.
- Cada cupón permite eliminar 1 día de atraso en la tarea recién entregada.

Posterior a la fecha de entrega de la tarea se abrirá un formulario de **Google Form**, en el cual deberán indicar si desean optar por una entrega atrasada y si harán uso de 0, 1 o 2 cupones para la entrega.

## 10. 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.

En 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.
- **Azul:** cada ítem en el que se evaluará con el archivo `cripto.py`.

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 Coordinador de Bienestar o al ayudante de Bienestar de tu sección, puedes hacerlo escribiéndoles a sus respectivos correos.

## 11. 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 3.10.
- 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 prohibida. 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. El no incluir este archivo o bien se encuentra vacío conllevaría un [\*\*descuento\*\*](#) en tu nota.
- Entregas con atraso entre 0 a 24 horas tendrán como calificación máxima 6,0. Entregas entre 24 y 48 horas tendrán calificación máxima 4,0. Más allá de este plazo, la calificación será mínima (1,0).
- Tendrás a tu disposición solo **dos** cupones en el semestre. Cada cupón permite eliminar 1 día de atraso en la tarea recién entregada.
- En ambas situaciones, ya sea entregar atrasado y/o utilizar 0, 1 o 2, deberán llenar el **formulario** correspondiente que será liberado una vez terminado el plazo oficial de la tarea.
- 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).