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

IIC2233 Programación Avanzada (2019-2)

Tarea 03

Entrega

- **Tarea**
  - **Fecha y hora:** sábado 07 de diciembre de 2019, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T03/`
- `README.md`
  - **Fecha y hora:** lunes 09 de diciembre de 2019, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T03/`

Objetivos

- Aplicar conocimientos de creación de redes para comunicación efectiva entre computadores.
- Diseñar e implementar una arquitectura cliente-servidor.
- Diseñar e implementar estructuras de datos propias basadas en nodos, para modelar y solucionar un problema en concreto.
- Aplicar conocimientos de manejo de *bytes* para seguir un protocolo de comunicación preestablecido.

Índice

1. *DCClub* **3**

2. Flujo del programa **3**

3. *Networking* **3**
- 3.1. Arquitectura cliente-servidor . . . 4
  - 3.1.1. Separación funcional . . . 4
  - 3.1.2. La conexión . . . 5
  - 3.1.3. Envío de mensajes . . . 5
  - 3.1.4. Ejemplo de codificación . . . 5
  - 3.1.5. *Logs* del servidor . . . 6
- 3.2. Roles . . . 6
  - 3.2.1. Servidor . . . 6
  - 3.2.2. Cliente . . . 7

4. Interfaz **7**
- 4.1. Ventana de inicio . . . 7
- 4.2. Ventana principal . . . 7
- 4.3. Ventana de chat . . . 8

5. Amistades **8**
- 5.1. Consultas . . . 10
  - 5.1.1. Ejemplos de consultas . . . 10

6. Archivos **11**
- 6.1. Archivos entregados . . . 11
  - 6.1.1. `usuarios.json` . . . 11
  - 6.1.2. `amistades.json` . . . 11
  - 6.1.3. `sprites` . . . 11
  - 6.1.4. `bonus` . . . 12
- 6.2. Archivos a crear . . . 12
  - 6.2.1. `parametros.json` . . . 12

7. *Bonus* **12**
- 7.1. Foto de perfil (2 décimas) . . . 13
- 7.2. Detección de consultas con RegEx (2 décimas) . . . 13
- 7.3. Palabras bobba (2 décimas) . . . 13
- 7.4. Movimiento de personajes (3 décimas) . . . 13
- 7.5. Mensajes con fotos (3 décimas) . . . 13
- 7.6. DCCMascota (3 décimas) . . . 14
- 7.7. DCCPalabra (3 Décimas) . . . 14
- 7.8. Servidor y Cliente Robusto (5 décimas) . . . 14

8. Avance de tarea **15**

9. `.gitignore` **15**

10. Importante: Corrección de la tarea **15**

11. Restricciones y alcances **16**

1. *DCClub*

Luego de años en el *DCCampo*, Enzo logra volver al DCC. Lamentablemente, ha pasado suficiente tiempo para que el malvado emperador EnZurg lograra convencer a todos que él era el verdadero jefe. Enzo, indignado, decide crear un plan para desterrar a su malvado gemelo y recuperar la paz en el DCC. Pero no será fácil, Enzo necesita nuevos aliados y una forma de comunicarse para no levantar sospechas.

Es por ello que Enzo decide crear una novedosa plataforma de comunicación, con el fin de conseguir secuaces y organizarse para poder derrotar al malvado EnZurg. Lamentablemente no puede hacerlo solo, ha pasado tanto tiempo que a Enzo se le olvidaron todos los lenguajes de programación. Por lo que te encomienda a ti, conocedor de `PyQt5`, *networking* y grafos, la misión de crear *DCClub*.

Figura 1: Logo de *DCClub*.

2. Flujo del programa

DCClub es una red social hecha con el fin de conectar a las personas que conforman el DCC. Cada usuario tiene un perfil y un personaje, y puede relacionarse con los otros usuarios de manera virtual, a través de un chat gráfico.

Al iniciar el programa, se abrirá la Ventana de inicio, donde se dará la opción de **ingresar** con un nombre de usuario preexistente, para así conectarse a la plataforma *DCClub*. El servidor debe ser capaz de identificar si el usuario existe y que no esté en línea, para poder permitir el ingreso a la plataforma.

Luego del inicio de sesión, se abrirá la Ventana principal, donde se presenta toda la información personal del usuario y además se darán las opciones de **unirse a una nueva sala de *chat***.

Una vez dentro de una sala, el usuario podrá **chatear** únicamente con los usuarios que están conectados e ingresaron a la misma sala. Además podrás realizar consultas a través del *chat*, las cuales serán respondidas por el personaje del servidor.

3. *Networking*

Para lograr la comunicación entre los distintos usuario del *DCClub*, deberás implementar una arquitectura **cliente-servidor**. Para esto, tendrás que hacer uso del *stack* de protocolos de red TCP/IP a través de la librería `socket`. Esta biblioteca te proporcionará las herramientas necesarias para administrar la conexión entre dos computadores conectados a una misma red local[1].

Es decir, debes implementar **dos** programas: el programa servidor y el programa cliente. El servidor es el primero que debe ejecutarse. Luego de empezar, queda a la espera de que distintos clientes simultáneos se conecten a él para acceder a sus funcionalidades. Es importante destacar que en el modelo cliente-servidor, la comunicación ocurre **exclusivamente** entre los clientes y un único servidor, es decir, nunca directamente entre dos clientes.

[1] Una red local es la interconexión de varias computadoras, puedes ver más información en **este artículo en Wikipedia (Red de área local)**.

3.1. Arquitectura cliente-servidor

En esta sección se presentarán consideraciones que deberás aplicar en la elaboración de tu tarea. Todo lo que se indique deberás seguirlo al pie de la letra, mientras que lo que no se encuentre especificado podrás implementarlo de la forma en que te sea más fácil mientras cumpla con lo mínimo pedido (siéntete libre de preguntar por este mínimo si no está especificado o no queda completamente claro).

3.1.1. Separación funcional

El cliente y el servidor deberán estar separados, lo que implica que deben estar contenidos en directorios diferentes, uno llamado `client`, y el otro llamado `server`. Cada uno debe tener un archivo llamado `main.py` (archivo principal a ejecutar) y debe contar con todos los módulos y demás archivos necesarios para su correcta ejecución. El siguiente diagrama ilustra la estructura descrita.

```
T03
├── client
│   ├── main.py
│   ├── parametros.json
│   └── ...
├── server
│   ├── main.py
│   ├── parametros.json
│   └── ...
├── .gitignore
├── README.md
└── ...
```

Aunque el cliente y el servidor compartan el mismo directorio padre (`T03`), esto es así solo para efectos de la entrega de la tarea y la ejecución de uno no podrá depender en absoluto de ningún archivo del otro, es decir, el cliente y el servidor deben estar implementados como si estuvieran en computadores diferentes. En la siguiente figura se muestra un diagrama que explica la separación esperada entre los distintos componentes de tus programas:

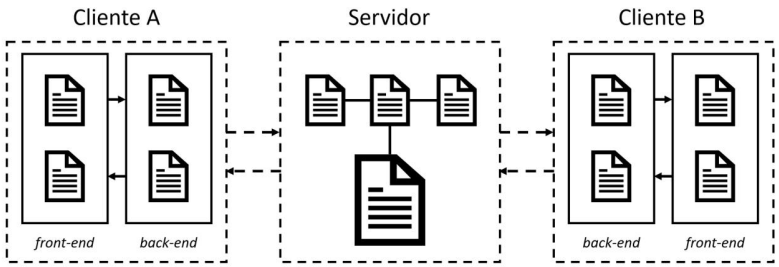

Figura 2: Separación cliente-servidor y *front-end-back-end*.

La comunicación entre el *back-end* y el *front-end* se debe realizar mediante señales, mientras que la comunicación entre el cliente y el servidor debe realizarse mediante *sockets*. Además unicamente los clientes presentarán una interfaz gráfica.

3.1.2. La conexión

El servidor abrirá un *socket* haciendo uso de los datos encontrados en el archivo `server/parametros.json`. Tal como sugiere su extensión, éste será de tipo JSON y seguirá el formato mostrado a continuación. Por otra parte, el cliente deberá conectarse al *socket* abierto por el servidor haciendo uso de los datos encontrados en `client/parametros.json`.

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

3.1.3. Envío de mensajes

Una vez establecida la conexión, los mensajes que se envíen entre cliente y servidor **deberán** codificarse siguiendo la siguiente estructura:

- Los primeros 4 *bytes* deben indicar el largo del contenido del mensaje, los que tendrán que estar en formato *little endian*[2].
- Luego, el contenido del mensaje debe ser dividido en tantos bloques de 124 *bytes* como sea necesario y, para cada uno, se deberán anteponer 4 *bytes* que indiquen el número del bloque, los que estarán codificados en *big endian*[2]. Si el último bloque resulta ser de largo menor a 124 *bytes*, se deberá rellenar con *bytes* 0 (`b'\x00'`) hasta alcanzar ese largo.
- Para transformar *strings* a *bytes* deberás usar UTF-8, así no tendrás problemas con las tildes ni caracteres especiales de ningún tipo.

Si no llegas a implementar esta funcionalidad en **Importante:** Corrección de la tarea se explica qué puedes hacer en dicha situación.

3.1.4. Ejemplo de codificación

Supongamos que queremos enviar un *string* largo como mensaje entre el cliente y servidor, tal que al codificarlo como *bytes* utilizando UTF-8 resulta en una cadena de 300 *bytes*.

Como especificado, para enviarlo es necesario separarlo en bloques de 124 *bytes*. Luego, para los 300 *bytes* son necesarios tres bloques. Los primeros dos se utilizan completos ($124 \times 2 = 248$) y su contenido proviene de la cadena original, pero para el tercero solo es necesario utilizar 52 *bytes* para lo restante, sobrando 72 *bytes*. Esos últimos *bytes* del tercer bloque son rellanados con ceros (`b'\x00'`).

Dada esa separación por bloques, se puede crear la codificación para ser enviada. Los primeros cuatro *bytes* son el largo del contenido del mensaje original (300 *bytes*) en *little endian*: `b'\x2c\x01\x00\x00'`. Luego, vienen los bloques de contenido de 124 *bytes* cada uno, cada uno precedido por 4 *bytes* que indican el número de bloque que sigue en *big endian*: para el bloque 1, `b'\x00\x00\x00\x01'`; para el bloque 2, `b'\x00\x00\x00\x02'`; y para el bloque 3, `b'\x00\x00\x00\x03'`.

En la siguiente figura se muestra gráficamente como está compuesta la codificación del ejemplo. En azul se muestran las porciones de bloques usadas por el contenido original del mensaje, mientras que en púrpura se muestra el rellenado con ceros para cumplir con el espacio de 124 *bytes* por bloque.

[2] El *Endianness* es el orden en el que se guardan los *bytes* que representan un número. Esto es relevante porque cuando intentes usar los métodos `int.from_bytes` e `int.to_bytes` deberá proporciones el *endianness* que debes usar, además de la cantidad de *bytes* que quieres usar para representarlo. Para más información puedes revisar este **enlace**.

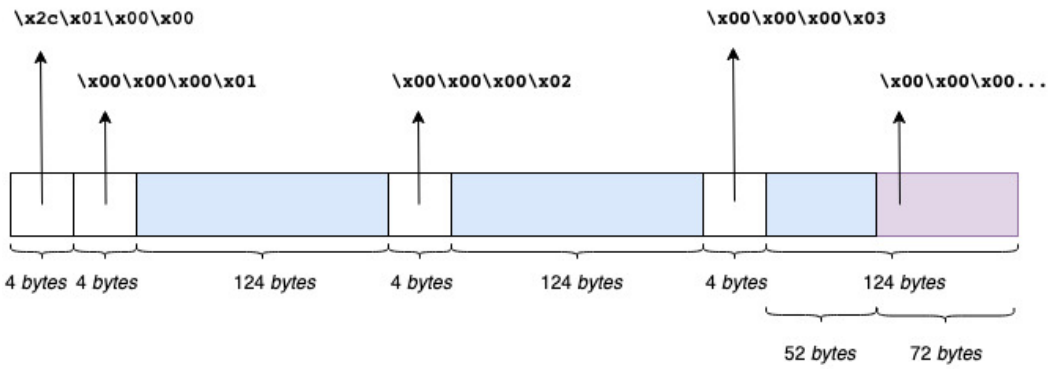

Figura 3: Ejemplo gráfico de codificación.

3.1.5. *Logs* del servidor

Como se indicó anteriormente, el servidor no cuenta con una interfaz gráfica. En su lugar, debe dejar registros de lo que ocurre en él constantemente mediante la consola. Estos se conocen comúnmente como *logs*, que para efectos prácticos se traducen en llamar a `print` cada vez que ocurra un suceso en el servidor. Específicamente, debe dejar un *log* cada vez que:

- Un cliente se conecte al o desconecte del servidor, identificando el cliente. Para identificarlo, se le debe asignar un nombre o *id* al cliente al conectarse.
- El cliente ingresa a o sale de una sala de chat. Se debe identificar el cliente y la sala en cuestión.
- El servidor reciba un paquete de información de algún cliente. Se debe indicar qué cliente envió el paquete, el tamaño del mensaje y su contenido decodificado.
- El servidor mande un mensaje a algún cliente. Debe indicar a qué cliente se el envía, el tamaño del mensaje y su contenido decodificado.

El formato para hacer los *logs* **queda a tu criterio mientras exponga toda la información de forma ordenada**, pero se recomienda un formato similar a:

```
 Cliente   |          Acción          |         Detalles
---------- | ------------------------ | ------------------------
   Enzo    |        Conectarse        |             -
   Enzo    |       Ingresar sala      |          Sala 1
   Enzo    |      Recibir Mensaje     |    7 bytes, "Hola :D"
  EnZurg   |        Conectarse        |             -
   Enzo    |       Desconectarse      |             -
```

La implementación de estos *logs* puede ser de mucha ayuda, ya que te permitirá comprobar el correcto funcionamiento del servidor y potencialmente te puede ayudar a encontrar *bugs* de funcionamiento.

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

- Es el encargado de **procesar y validar** las acciones realizadas por todos los clientes. Por ejemplo: procesar los mensajes de chat y las consultas enviadas al servidor.

- Debe **distribuir los cambios** en tiempo real a los clientes conectados correspondientes, con el fin de mantener sus interfaces actualizadas y consistentes entre sí. Por ejemplo: cuando un usuario entra o sale de una sala de chat, se debe actualizar la información en Ventana principal como en la Ventana de chat.
- Deberá **almacenar y actualizar** toda la información de los usuarios y sus amistades. Por ejemplo: cuando un usuario agrega o elimina un amigo.
- No tiene interfaz gráfica asociada, pero debe tener un buen registro de *logs* en consola.

3.2.2. Cliente

- **Envía las acciones realizadas** por el usuario al servidor.
- **Recibe e interpreta** las respuestas y actualizaciones que envía el servidor.
- Tiene una interfaz gráfica asociada, que deberá permitir la **interacción del usuario con el programa** y deberá mantenerse **actualizada en todo momento** con la información que se recibe del servidor.

4. Interfaz

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

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

4.1. Ventana de inicio

En esta ventana el usuario puede iniciar sesión e ingresar a la plataforma, para esto la ventana debe presentar un cuadro de texto para ingresar el nombre de usuario y un botón para iniciar la sesión.

El usuario logrará iniciar sesión si cumple las siguientes condiciones:

- Su nombre de usuario pertenece a los usuarios registrados, estos los puedes encontrar en el archivo `usuarios.json`.
- El usuario actualmente no se encuentra conectado.

Si se presiona el botón de ingreso cumpliendo las condiciones, el usuario se lleva a la Ventana principal.

4.2. Ventana principal

Una vez que el usuario inicie sesión, se deberá encontrar con una nueva ventana que muestre la información personal del usuario: nombre de usuario, cantidad de amigos y la imagen de su personaje; además de un botón[3] que le permita al usuario cerrar la sesión y volver a la Ventana de inicio.

Además, *DCClub* debe ser capaz de manejar múltiples conversaciones a la vez, para lo que deberás implementar **salas de chat**: espacios donde hasta cinco usuarios pueden enviarse mensajes. Desde Ventana

[3]El botón que incluyen las ventanas `(X)` también es válido, para configurarlo debes sobrescribir el método `closeEvent(event)` de la ventana. Si lo implementas deben indicarlo en tu `README`.

principal el usuario podrá visualizar todas las salas de manera clara. Para cada sala se debe mostrar la **cantidad de usuarios** que han ingresado a ella y el **nombre de esta**.

La aplicación tendrá **cuatro salas de chat** activas, las cuales siempre existirán a pesar de estar vacías. Los nombres de estas salas quedan a tu criterio mientras sean únicos entre ellos.

Toda esta información debe ser fácilmente accesible y tiene que ser actualizada en tiempo real. Por ejemplo, si una persona entra o sale de una sala, a todos los usuarios que se encuentran conectados al servidor se les debe actualizar el valor de usuarios conectados dicha la sala.

Naturalmente, cada sala deberá contar con la opción de **unirse** a ella. En caso de que el usuario se una a una sala, el programa mostrará la Ventana de chat y el usuario hará ingreso a ella. Si la sala está llena, es decir, que el número de usuarios que la componen haya llegado a su máximo de **cinco usuarios**, no debe permitir al usuario ingresar.

4.3. Ventana de chat

Una vez que el usuario ingrese a una sala de chat se deberá ver el personaje del usuario junto a los personajes de los otros usuarios y un personaje que represente al servidor. Además deberá existir un cuadro de texto para ingresar los mensajes y un botón para poder enviarlos.

A medida que los distintos usuarios ingresen mensajes, todos los usuarios conectados a la sala deberán ver los mensajes sobre la cabeza del personaje correspondiente, a continuación se muestra un ejemplo de como se debe ver:

Figura 4: Ejemplo de visualización del *chat* entre dos usuarios.

El personaje del servidor se encargara de mostrar las respuestas de las consultas que se le hagan a este, las consultas se explican con mayor detallen en Consultas.

Puedes posicionar a los personajes y determinar la forma en que se vean los globos de diálogo de la forma en que te sea más sencillo, mientras se puedan ver todos los personajes, sus respectivos globos de texto y estos sean ***scrolleables***[4].

Finalmente, cada vez que un usuario o el servidor mande un nuevo mensaje, queda a tu criterio si reemplazar el mensaje que se estaba mostrado previamente en el globo de texto, o de alguna forma crear múltiples globos ordenados que muestren distintos mensajes.

5. Amistades

En tu programa los usuarios además de poder interactuar dentro del chat, pueden relacionarse al declararse como **amigos**. Los amigos se comportarán de **casi** igual forma que las que estamos acostumbrados a ver

[4]Para esto te recomendamos utilizar la clase `QTextEdit`, ya que automáticamente hace el texto *scrolleable* con respecto a la geometría del *widget*. Para evitar que el texto se pueda editar desde la interfaz, debes utilizar el método `setReadOnly(bool)`.

en redes sociales:

- Una persona podrá amistar[5] a otra y desde ese momento serán amigos, sin necesidad de la aceptación de la otra persona. Es decir, las amistades funcionan sin solicitud.
- Como esto puede generar incomodidades a los usuarios, también existe la posibilidad de eliminar amistades.

Las interacciones (amistades) entre los usuarios deben modelarse como estructura de **grafos no-dirigidos**, es decir, los usuarios se entenderán como nodos y cada amistad es una arista no-dirigida entre ellos.

Además, deberás implementar las siguientes funcionalidades:

- **Alcance de amistad:**[6] para cierto usuario y distancia, se define como la lista de usuarios que se encuentran a tal distancia exacta mediante la relación de amistad. Por ejemplo, los amigos de los amigos de A, que no son amigos de A, son parte del alcance de amistad a distancia dos de A.
- **Afinidad entre dos usuarios:** es una medida numérica que se calcula de la siguiente forma para dos usuarios a y b:

$$afinidad = \frac{\#\text{ total de amigos en común entre } a \text{ y } b}{\#\text{ total de amigos de } a \text{ o } b}$$

  En caso de que alguno de los dos usuarios no tenga amigos, la afinidad entre ellos será 0.
- **Recomendación de amigos:** se entregarán todos los usuarios que se encuentren a distancia dos a partir del usuario, es decir, se recomendarán todos los amigos de sus amigos que no pertenezcan al grupo de amigos del usuario.

A continuación un ejemplo de un grafo y el resultado de algunas consultas:

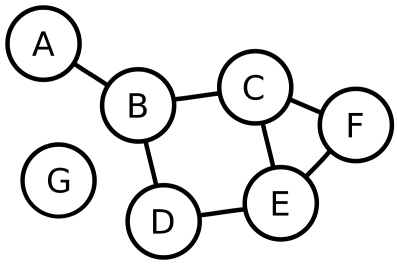

Figura 5: Grafo de amistad entre los usuarios A, B, C, D, E, F y G.

- El alcance de amistad de A a distancia 3 son E y F.
- La afinidad entre B y E es $1/2$, ya que tienes 2 amigos en común (C y D) y 4 usuarios que son amigos de B o E (A, C, D y F).
- La afinidad entre C y G es 0, ya que G no tiene amigos.
- La recomendación de amigos de F serían: B y D, ya que son amigos de C y E.
- La recomendación de amigos de G no entrega resultados, ya que no tiene amigos a distancia 1.

[5] Amistar (RAE).
[6] Esta funcionalidad te puede ser de ayuda para las siguientes.

5.1. Consultas

Mediante las salas de chat es posible hacer consultas sobre el grafo de amistades de *DCClub* para acceder a las anterior funcionalidades. Ingresando ciertos comandos especiales en el chat, el servidor debe identificarlos y responder mediante su personaje los resultados a la consulta asociada.

Deberás implementar los siguientes comandos:

- `/friend <nombre usuario 1> <nombre usuario 2>`: hace que dos usuarios sean amigos. Debes actualizar el archivo que almacena el grafo en ese momento.
- `/unfriend <nombre usuario 1> <nombre usuario 2>`: hace que dos usuarios dejen de ser amigos. Debes actualizar el archivo que almacena el grafo en ese momento.
- `/get reachable <nombre usuario> <distancia>`: retorna el alcance de amistad de un usuario a cierta distancia.
- `/get affinity <nombre usuario 1> <nombre usuario 2>`: retorna la afinidad entre dos usuarios.
- `/get recommendation <nombre usuario 1>`: retorna la lista de recomendación de amigos de un usuario.

El campo de texto para enviar la consulta **es el mismo que el usado para hablar por el chat**. El programa del **cliente** debe identificar si un mensaje es un comando o no, y en caso de ser un comando debe identificar que esté correctamente escrito. Los nombres de los usuarios pueden estar compuestos por más de una palabra, pero siempre se escribirá entre comillas simples.

Luego, el programa del **servidor** debe recibir tal comando, interpretarlo, procesarlo y responder con un resultado. **Este programa debe calcular el resultado de una consulta en el momento en que se pide**: precalcular los resultados puede ser negativo para tu programa ya que eventualmente significa mucha carga de computación innecesaria.

Los resultados de comandos solo los recibe el usuario que los accionó. Y, para todos los comandos se deberá informar si se ejecutó correctamente el comando o si hubo un error. Por ejemplo, en el caso de que uno de los usuarios especificados en el comando no existe, debe informarse que no se pudo completar el comando.

Finalmente, si no llegas a integrar las consultas, pero lograste implementar las distintas funcionalidades pedidas, en **Importante:** Corrección de la tarea se especifica que puedes hacer en dicha situación.

5.1.1. Ejemplos de consultas

Las siguientes son respuestas válidas a ciertos comandos, siguiendo el ejemplo anterior de amistades.

1. `/friend 'A' 'G'` ⟶ `"Amistad creada correctamente entre A y G"`
2. `/unfriend 'A' 'G'` ⟶ `"Amistad eliminada correctamente entre A y G"`
3. `/friend 'A' 'Z'` ⟶ `"Error: usuario no existe"`
4. `/get reachable 'E' 1` ⟶ `"Alcance E distancia 1: D, C, F"`
5. `/get reachable 'A' 3` ⟶ `"Alcance de A distancia 3: E, F"`
6. `/get affinity 'B' 'E'` ⟶ `"Afinidad entre B y E: 0.5"`
7. `/get affinity 'B' 'Z'` ⟶ `"Error: usuario no existe"`
8. `/add recommendation 'F'` ⟶ `"Recomendación de amigos para F: B y D."`

6. Archivos

Para desarrollar tu programa de manera correcta deberás hacer uso de los siguientes archivos:

6.1. Archivos entregados

Se te entregarán una serie de archivos que te ayudarán a la implementación de tu tarea.

6.1.1. `usuarios.json`

Almacena la información de todos los usuarios de la aplicación. Contiene el nombre de usuario y el tipo de personaje del usuario.

Ejemplo del contenido del archivo:

```
[
    {
        "nombre": "Enzini",
        "personaje": "Robot"
    },
    ...
]
```

6.1.2. `amistades.json`

Almacena todas las relaciones de amistad del programa. Cada nombre de usuario corresponde a una llave y tiene asociada la lista con los nombres de sus amigos. En caso de no tener amigos, la lista estará vacía.

Ejemplo del contenido del archivo:

```
{
    "gatochico": [
        "terminator",
        "JavOchito",
        "Kawatron3_0"
    ],
    ...
}
```

6.1.3. `sprites`

Se te entregará una carpeta llamada `sprites`, la cual contendrá todas las imágenes necesarias para implementar tu programa. Esta contiene los siguientes directorios:

- `personajes`: contiene las distintas apariencias y poses de los personajes de los usuarios.

- `fondos`: contiene diversos fondos para las salas de chat del programa.

Se te entrega una gran variedad de imágenes para que tengas mayor libertad al momento de diseñar la aplicación, no es necesario que utilices todas las imagenes entregadas. También puedes utilizar tus propias *sprites*, pero recuerda deberás guardarlas en otra carpeta, ya que **no debes** modificar el contenido de la carpeta `sprites`.

6.1.4. `bonus`

Por último, se te entregara una carpeta llamada `bonus`, la que contendrá todos los archivos necesarios para realizar los bonus que se explicarán más adelante. En el caso de que quieras utilizar tus propias *sprites*, al igual que el caso deberás guardarlo en otra carpeta, ya que no debes modificar el contenido de estos archivos.

6.2. Archivos a crear

6.2.1. `parametros.json`

Similar a las tareas anteriores, deberás crear y rellenar un archivo que contenga **todos** los parámetros, constantes, *paths*, entre otros, que necesiten tus programas para funcionar. En esta ocasión el archivo deberá ser un `parametros.json`, donde cada llave del archivo tendrá asociado. Un ejemplo de su formato sería el siguiente:

```
{
    "cantidad de salas": 4,
    "cantidad usuario salas": 5,
    ...
}
```

Como se indica en Separación funcional, deberás crear un archivo `parametros.json` para el servidor y otro para el cliente, ya que estos deben contener los datos necesarios para el funcionamiento de su respectivo programa.

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**[7].
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 **10 décimas**.

[7] Esta nota es sin considerar posibles descuentos.

7.1. Foto de perfil (2 décimas)

Para obtener este *bonus*, deberás implementar el botón **Foto Perfil** en la ventana principal, a través del cual el usuario podrá seleccionar una imagen y subirla como foto de perfil. Esta imagen deberá aparecer junto a la información personal del usuario. La foto debe quedar guardada, de modo que cuando el usuario vuelva a conectarse ésta se muestre automáticamente.

7.2. Detección de consultas con RegEx (2 décimas)

Este *bonus* busca evaluar el uso de expresiones regulares para identificar las consultas señaladas en Consultas. Para esto deberás generar **una** expresión regular que indique si un texto ingresado cumple con la sintaxis esperada o no.

7.3. Palabras bobba (2 décimas)

En el *DCClub* existe un conjunto de palabras prohibidas que queremos evitar que se mencionen, por lo tanto para obtener este *bonus*, cada vez que aparezcan en el mensaje de algún usuario estas serán censuradas. Para censurar las palabras tendrás que sustituirlas por una serie de caracteres del mismo largo que estará formado por *random* entre los caracteres `!?#$%&*_@¡¿`.

Para saber cuales son las palabras bobba, debes buscarlas en el archivo `palabras_bobba.txt`, este archivo contiene todas las palabras prohibidas para los usuarios, donde cada línea corresponde a una palabra prohibida escrita en minúsculas. A continuación un ejemplo del contenido del archivo:

```
dcc
enzurg
saiding
...
```

Estas palabras deben ser censuradas sin cambiar las mayusculas y minusculas de las palabras que no estan prohibidas.

7.4. Movimiento de personajes (3 décimas)

Tal como dice el nombre de este *bonus*, deberás implementar el movimiento de cada personaje en la sala. Esto implica que los todos los jugadores podrán ver cuando alguien se mueve.

Su movimiento será lateral, usando las teclas `A` y `D` para moverse hacia la izquierda y derecha respectivamente. Los jugadores podrán colisionar entre ellos y con el borde del mapa, así que debes considerar que al entrar a una sala no aparezca uno encima de otro. Además, al moverse el personaje deberá moverse con él su globo de texto. Finalmente, deben avanzar una velocidad constante en píxeles y sus *sprites* deben actualizarse.

7.5. Mensajes con fotos (3 décimas)

La idea de este *bonus* es que los usuarios de *DCClub* puedan subir imágenes a través de los globos de dialogo que tienen arriba de sus avatares. Para obtener este *bonus* debes implementar un botón **Subir Imagen**, junto al botón de enviar mensajes, que abrirá un selector de imágenes, en donde puedas buscar y subir una imagen de tu computador.

La imagen cargada debe aparecer en el globo de texto del avatar correspondiente como un mensaje más. Y finalmente, también debe tener un tamaño máximo permitido, y en caso de excederlo, debe ser escalada adecuadamente.

7.6. DCCMascota (3 décimas)

Este *bonus*, permite que los usuarios de *DCClub* puedan adoptar una mascota. Para obtener las décimas, deberás implementar el botón **DCCMascota**, que haga que automáticamente aparezca una mascota al lado del avatar del usuario, las *sprites* se encuentran en la carpeta `mascotas`.

Además cada mascota podrá comunicarse a través de sonidos, los cuales deberás buscar y descargar[8]. Cada vez que su dueño haga *click* sobre la mascota y esta gruñirá, maullará, piará, etc, según el animal que sea, permitiendo que todos los usuarios de la sala de chat escuchen el sonido de la mascota. A continuación se muestra un ejemplo de como se debe ver:

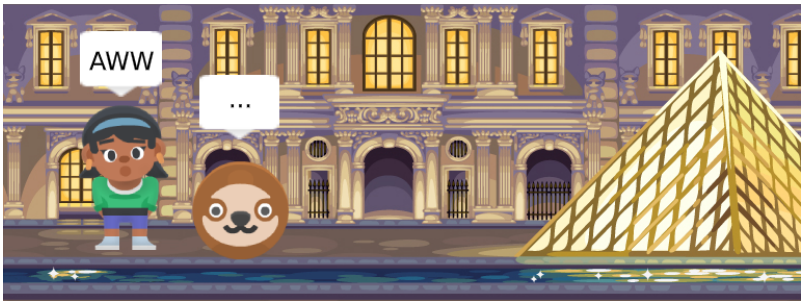

Figura 6: Ejemplo de visualización de usuario con su mascota.

7.7. DCCPalabra (3 Décimas)

Para obtener este *bonus*, deberás implementar un minijuego en donde los usuarios en una sala podrán adivinar una palabra escogida por el servidor. El juego debe tener como mínimo:

- Un comando enviable por cualquiera de los usuarios para comenzar el juego. Si un juego ya fue iniciado, el comando no tendrá efecto y se notificará mediante un mensaje del servidor.
- Un mensaje enviado por el servidor declarando el inicio del juego y mostrando una pista sobre la palabra.
- Un mensaje anunciando el ganador del juego una vez que se adivine la palabra correcta o el fin del juego cuando se llegue a un límite de tiempo.

Deberás agregar a tus parámetros las palabras posibles junto con una pista de cada una y el límite de tiempo del minijuego.

7.8. Servidor y Cliente Robusto (5 décimas)

Una buena aplicación distribuida es robusta ante la caída/desconexión de partes del sistema, es por eso que para obtener este *bonus* deberás lograr que tanto tu cliente como tu servidor sean robustos. Esto quiere decir, que si uno de los dos programas deja de funcionar (ya sea debido a pérdida de internet, corte eléctrico, etc.) el otro maneje correctamente la situación.

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

En caso de que un cliente deje de funcionar, el servidor debe detectar lo sucedido y seguir funcionando normalmente. Esto implica liberar toda conexión o rastro que se tenía de ella en el servidor, como dejar de escuchar el *socket* compartido y eliminar al usuario de cualquier sala de chat. También debe mostrar un mensaje en consola indicando `'El usuario X se desconectó'`, donde `X` es el nombre del usuario.

[8] En el siguiente ***link*** hay una gran variedad de sonidos de animales.

Para facilitar el testeo de esta funcionalidad, pueden probar realizando *keyboard interrupt* en la consola del cliente o del servidor. También pueden probar cerrando la interfaz de un cliente.

Se espera que puedan manejar un flujo alternativo de salida o control de la situación. No se considera suficiente manejar todos los casos mediante una sola línea de flujo, como mediante:

```
try:
    # ...
except ConnectionError:
    # ...
    exit()
```

Para salir, el programa debe seguir su curso natural y no hacerlo de manera forzada. Esto significa que debes permitir que las funciones llamadas retornen y los ciclos terminen. Debes cerrar las ventanas por tu cuenta y, de usar *threads* o *timers*, estos **deben** ser *daemons*.

8. Avance de tarea

A diferencia de las tareas pasadas, esta tarea no incluirá una sección de avance. En su reemplazo, se aumentó el máximo de décimas que se pueden obtener a partir de los *bonus*, lo que cubre las potenciales décimas del avance.

9. `.gitignore`

Para esta tarea **deberás utilizar un** `.gitignore` para ignorar los archivos entregados, este deberá estar dentro de tu carpeta `T03/`. Puedes encontrar un ejemplo de `.gitignore` en el siguiente ***link***.

Deberás ignorar el enunciado, todos los archivos entregados en las carpetas `sprites` y `bonus` y los archivos: `usuarios.json` y `amistades.json`.

En caso de que hagas uso de tus propias *sprites*, deberás guardarlas en otra carpeta y asegurarte que no sean ignoradas por el `.gitignore`.

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

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

10. Importante: Corrección de la tarea

Para esta tarea, el carácter funcional del juego 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.
- **Azul**: cada ítem en el que se evaluará el correcto uso de señales.

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

Se recomienda el uso de *logs* (*Logs* del servidor) para ver los estados del sistema para apoyar la corrección. De todas formas, ningún ítem de corrección se corrige puramente por consola, ya que esto no valida que un resultado sea correcto necesariamente. Para el cliente, todo debe verse reflejado en la interfaz, pero el uso de *logs* también es recomendado.

En caso de que no logres completar el Envío de mensajes, puedes enviar los mensajes mediante codificación directa, para que así no afecte otras funcionalidades de la tarea. Pero implicará que no obtengas puntaje en los ítems relacionados. Si lo llegases a implementar de esta forma, recuerda indicarlo en tu `README`.

Finalmente, en caso de que logres implementar las consultas del grafo de amistad, pero no logres integrarlo en tu programa, puedes dejar un archivo `.py` que permita comprobar la lectura y guardado tanto del grafo como todas las consultas indicadas en Consultas. Si llegases a entregar este archivo `.py`, recuerda indicar su nombre en tu `README`.

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 3.7.
- 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 48 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).