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

**IIC2233 - Programación Avanzada (I/2017)**
**Tarea 6**

# 1. Objetivos

- Aplicar conceptos y nociones de networking para la creación de un servicio en linea.
- Aplicar conceptos del serialización para el manejo de envio de archivos.
- Aplicar conocimientos de interfaces.

# 2. Introducción

Después de pasar un rato jugando entre compañeros, te diste cuenta que tus habilidades para **League of Progra** no son las mejores. Luego de estar un rato enojado por perder tantas partidas, se te ocurrió una brillante idea: crear un juego donde ganes siempre a cada contrincante. Esto no es una tarea fácil pero sabes que tu conocimiento musical es superior al de los otros alumnos así que tras unas horas de analizar el problema concretaste tu idea: PrograPop.

# 3. PrograPop

**ProgaPop** es un juego online donde los jugadores deben adivinar canciones. Para participar, el jugador debe escoger una sala de juego (Pop, Rock, 80s, ...), escuchar la canción que están tocando y hacer click en el cantante o título de la canción lo más rápido posible [1].

Para que su aplicación sea un éxito usted deberá crear un servidor que manejará el estado global de la aplicación, así como también coordinar las acciones que pueden ejecutar los clientes. Además deberá implementar los clientes que son programas que se conectan e interactúan con el servidor.

## 3.1. Ingreso

Al ingresar al sistema el jugador debe ingresar su nombre. No se debe permitir ingresar a usuarios con el mismo nombre. El sistema debe cargar su puntaje actual en la interfaz. Si ingresa por primera vez, su puntaje es cero. Cuando entre a la aplicación el usuario debe poder ver las diferentes salas y la tabla de puntajes en la *pantalla de inicio*.

En la *pantalla de inicio* se debe mostrar la siguiente información de las salas:

- Nombre del usuario.
- Cantidad de jugadores presentes en la sala.

[1] Pueden ver una vil casi copia de su juego en https://www.youtube.com/watch?v=M0YIe9GLGE8

- Segundos restantes de la canción que actualmente se está reproduciendo.
- Nombres de al menos 2 artistas que se reproducen en la sala.

## 3.2. Salas de Juegos

Las salas de juegos tienen canciones de un mismo estilo. Los archivos de las canciones se encuentran en formato WAV (extensión `*.wav`). Éstos se pueden reproducir en el programa mediante la clase `QSound` perteneciente a `PyQt5`.

Cada canción se reproduce por un periodo de 20 segundos y luego se pasa a la siguiente, incluso si todos los jugadores presentes en la sala ya adivinaron la canción. Los usuarios deberán adivinar el título de la canción o artista. Siempre que haya jugadores en la sala, se debe estar tocando una canción. La pregunta que los usuarios deben responder debe ser aleatoria.

Cuándo un usuario se encuentre al interior de una sala, la interfaz gráfica debe mostrar lo siguiente::

- Las posibles respuestas para la pregunta actual.
- La *tabla de los tiempos* que demoraron los usuarios en contestar la pregunta de la canción.
- Los jugadores descalificados para la canción actual (que no respondieron correctamente).
- El tiempo restante de la canción actual.
- Los nombres y puntajes actuales de los jugadores de la sala.

Los jugadores pueden entrar en cualquier momento a la sala de juegos. Si la sala está tocando una canción, el usuario debe esperar hasta la siguiente canción para poder jugar, pero debe poder ver la *tabla de tiempos* de la canción actual. Así mismo, pese a no jugar, debe poder escuchar la canción sincronizada con el resto de los jugadores.

### 3.2.1. Salas especiales

Existirá una versión especial de cada una de las salas. Estas versiones tendrán las mismas canciones pero con filtros para que sea más difícil reconocerlas. La versión especial debe aparecer como otra sala más, pero su nombre debe ser "{Nombre original de la sala}-{Nombre de filtro}".

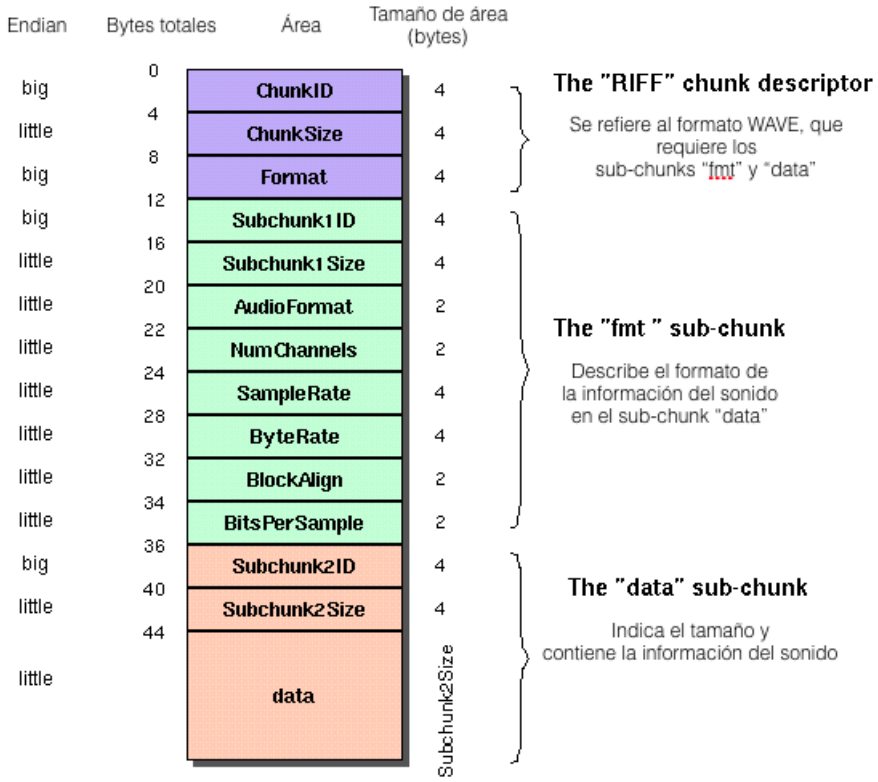
## 3.3. Canciones

Cada canción[2] se encontrará en formato WAV (con nombre `Artista - Nombre de canción.wav`), dentro de una carpeta con el nombre de la sala a la cuál pertenece.La estructura de este tipo de archivos es la siguiente: un *header* compuesto de 44 bytes y luego *chunks* con la información de la canción. A continuación se presenta una tabla con el detalle de la estructura de un archivo WAV:

[2]Puede encontrar canciones de muestra en el siguiente enlace: `https://drive.google.com/drive/folders/0ByciveqMYys_WjBLTkNDRGVfYXM`.

Cuadro 1: Área - Descripción

| | |
|---|---|
| ItemChunkID | ASCII string "`RIFF`" |
| ChunkSize | Tamaño total del archivo |
| Format | ASCII string "`WAVE`" |
| SubchunkID | ASCII string "`fmt `", con un espacio al final |
| Subchunk1Size | Cantidad de bytes anteriores: 16, 18 o 40 |
| AudioFormat | Formato en que se almacenan datos, PCM es formato mas común |
| NumChannels | 1 para sonido Mono, 2 para Stereo |
| SampleRate | Número de *samples* por segundo para cada canal, se usa generalmente 44100 para buena calidad de sonido |
| ByteRate | Número de bytes requeridos por segundo de audio data |
| BlockAlign | Número de bytes requeridos para almacenar un *sample* para cada canal |
| BitsPerSample | Número de bits por sample: 8, 16, 32 |
| Subchunk2ID | Marca el inicio de la sección data |
| Subchunk2Size | Indica el tamaño de data |




En la tabla se señala que ciertos segmentos son *big endian* o *little endian*. El *endianness* de un segmento describe el orden en que se almacenan en memoria los bytes. *Big endian* quiere decir que las cifras más significativas se almacenan primero. Por otra parte, *little endian* quiere decir que las cifras menos significativas se almacenan primero. De esta manera el byte `0b10100100` representa el valor 164 en *big endian* y 37 en

*little endian*.

El contenido (data) está compuesto por una cantidad de *samples* determinada en el *header*. Cada uno de ellos contiene *chunks* correspondientes a la cantidad de canales de audio del archivo. Si es un sonido *Mono* existe solo un canal y para sonido *Stereo* existen dos canales.

Para esta tarea se considerarán solo archivos WAV con sonido *Stereo*, por lo que trabajarán con una estructura de 2 canales, con cada *sample* de 4 bytes. Los *chunks* se ordenan de manera intercalada entre canales. De esta forma, los *chunks* pares son para un canal y los impares son para el otro canal.

### 3.3.1. Filtros

A continuación se describen dos filtros: *Ecualizador* y *2x Canción*. Para cumplir los objetivos de esta tarea sólo necesita implementar el filtro *Ecualizador*. La implementación del filtro *2x Canción* será bonificada con un aumento de 5 décimas sobre la nota obtenida.

- **Ecualizador**: la canción tiene que poder cambiar su frecuencia, para esto debe recibir dos parametros, el multiplicador de frecuencia (f) y el orden del filtro (n). La frecuencia resultante puede ser mayor o menor y esta depende del valor de f, ue al igual que n es elegido aleatoriamente por el servidor donde f es un numero decimal entre 0.5 y 3 y n en un numero entero entre 1 y 10. Para aplicar este filtro debe seguir dos pasos.

  1. Editar el header para que represente correctamente el cambio de frecuencia
  2. Editar el segmento data, agregando el filtro.

  - En el header, la sección de SampleRate representa la frecuencia de la canción, es necesario multiplicar esa frecuencia por f que representará que tanto aumentará o bajará la frecuencia.
  - En el data, como hay dos canales, es necesaria dividir la información por canal. Los chunks pares van a un canal y los impares al otro.
  - Luego, a cada canal de por sí se aplica el filtro, que consiste en un nuevo canal donde cada chunk se rige bajo la siguiente formula:

    $$chunk\_nuevo_i = \frac{chunk\_antiguo_i + chunk\_nuevo_{i-1}}{2}$$

    Donde $chunk\_antiguo_i$ representa el i-ésimo chunk del canal a filtrar y $chunk\_nuevo_i$ representa el i-ésimo chunk del canal resultante del filtro.[3]
  - Este último paso se repite n cantidad de veces
  - Finalmente se vuelve a juntar la sección de data en el mismo orden que en un principio, pero con los nuevos canales, juntando esto con el nuevo header.

- **2x Canción (Bonus)**: La canción tiene que poder escucharse al doble de la velocidad normal. Para hacer esto es necesario:

  - Extraer desde el archivo wav información necesaria para luego procesar: el número de bits por muestra (BitsPerSample), la frecuencia de muestreo (SampleRate) y el número de canales (NumChannels).
  - Se convierte el número de bits a número de bytes.
  - Para cada canal existente, se forma un arreglo en que cada elemento de éste son secciones del área data, donde el tamaño de cada sección viene dado por el número de bytes por muestra (N). El arreglo resultante será del tipo *[Data(desde 0 a N bytes), Data(desde N a N+1 bytes, .., ..]*

[3] En caso de que i sea 0 considerar i - 1 = 0

- Teniendo el arreglo con todas las muestras del wav, clasificadas por canal, luego solo resta modificar en el nuevo archivo a generar el número de samples que habrá en un segundo (SampleRate). Efectivamente, doblando el número de samples que se reproducen por segundo, el tiempo de reproducción disminuye a la mitad con la consecuente percepción de que la velocidad ha aumentado.
- Los headers y demás áreas (AudioFormat, etc) se mantienen iguales y se guardan en un nuevo archivo.

### 3.4. Puntaje

El puntaje que un jugador obtiene para una canción es el siguiente:

$$P = A * (20 - T) * 100$$

donde $P$ es el puntaje de esa canción, $A$ es igual a 1 si el jugador acierta o 0 en otro caso, $T$ es el tiempo en que demora en adivinar en segundos. Si un jugador se equivoca no puede volver a intentarlo.

Los usuarios pueden acceder a la tabla de puntajes de todos los jugadores en la sección puntajes de la pantalla inicial. Esta sección debe mostrar:

1. Todos los jugadores ordenados según sus puntajes
2. La sala en la que han contestado correctamente a más canciones
3. La sala en la que no han contestado o han contestado incorrectamente más canciones

## 4. Cliente-Servidor

Su implementación de PrograPop deberá basarse en la arquitectura *cliente-servidor*, en donde todas las interacciones que se quieran realizar entre usuarios deberán pasar por el servidor. Para lograr lo anterior, deberá implementar un protocolo eficiente de comunicación entre el cliente y el servidor usando el modelo **TCP/IP**.

Su sistema deberá ser capaz de funcionar entre computadores que estén conectados a una misma red local. Para esto es necesario que maneje de forma correcta los puertos y el host asignado a sus sockets. Para que esto funcione correctamente, y para que los ayudantes después sean capaces de corregir la tarea, **deberá tener los siguientes programas**.

### 4.1. Servidor

Este programa no necesita una interfaz gráfica, pues solo manejará la lógica del sistema y se ejecutará solo una instancia de él (en un único computador). En este computador es donde usted guardará los archivos "subidos" al sistema, la base de datos de los usuarios y todo lo que usted encuentre pertinente. También es de suma importancia que en el archivo en el que programe el servidor usted escriba las siguientes variables en las primeras líneas: `HOST` y `PORT`. Estas variables deben tener los valores del host y puerto que usa su programa, y deben ser usadas en el método `bind` del socket del servidor. De esta forma cuando el ayudante corrija su tarea, él o ella podrán escribir su propia IP y el puerto que desee utilizar como servidor. Para poder realizar una conexión por red local usted solo debe hacer dos cosa

1. Estar conectado a una red
2. Setear la variable `HOST` de su servidor como la IP local de su computador. Distintos sistemas operativos tienen distintas formas de obtenerla, pero en todos es un proceso muy simple.

### 4.2. Cliente

Este es el programa con el que interactuarán los usuarios, por lo que debe incluir la interfaz gráfica de `PrograPop`. El cliente debe ser capaz de contactar a un servidor para poder funcionar, el cual puede estar ejecutándose en el mismo computador, o en un remoto. Si el cliente fuera ejecutado en dos computadores distintos en la misma red ambos deberían ser capaces de probar el sistema interactuando con el servidor.

Al igual que en el caso del servidor, el archivo en el que programe el cliente deberá escribir las siguientes variables en las primeras líneas: `HOST` y `PORT`. Estas variables deben tener los valores del host y puerto que usa su programa, y deben ser usadas en el método `bind` del socket del servidor. De esta forma cuando el ayudante corrija su tarea, él o ella podrán escribir su propia IP y el puerto que desee utilizar como servidor.

### 4.3. Estructura de Archivos

La tarea se debe entregar con la siguiente estructura de carpetas y archivos:

```
T06
|-- client
|   `-- main.py
|   `-- (otros módulos del cliente)
`-- server
    |-- main.py
    |-- (otros módulos del servidor)
    `-- songs
        |-- Indie Rock
        |   |-- Snow Patrol - Chasing Cars.wav
        |   |-- The 1975 - Girls.wav
        |   |-- The Killers - Mr. Brightside.wav
        |   `-- The La's - There She Goes.wav
        `-- Pop
            |-- Bruno Mars-24KMagic.wav
            |-- Justin Bieber-Sorry.wav
            |-- The Chainsmokers-Closer.wav
            `-- The Weeknd-Starboy.wav

5 directories, 10 files
```

Profundizando la estructura anteriormente indicada, los programas del cliente y servidor deben estar separados, cada uno en su propia carpeta. Además, se puede observar como las canciones se encuentran dentro de la carpeta `songs` en la carpeta del servidor.

Los clientes y servidores deberán definir las constantes `HOST` y `PORT` en las primeras 2 líneas de su código.

## 5. Interfaz gráfica

Todas las interacciones que se esperen del cliente deben realizarse a través de interfaces gráficas. La interfaz debe ser amigable y enfocada en la usabilidad, es decir, comportarse de acuerdo a lo que espera el usuario de forma natural, sin eventos inesperados.

## 6. Bonificaciones

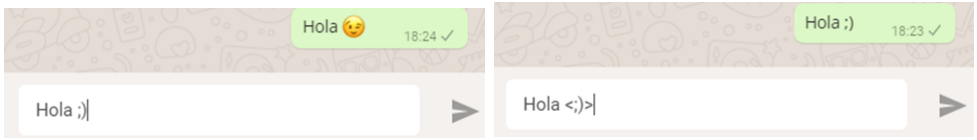
Esta tarea tiene tres bonus. El primero es el filtro *2x Canción* (sección 3.2.1), con el cual puedes aumentar tu nota en 5 décimas. Los otros bonus se explican a continuación, junto con la cantidad de décimas en que puedes aumentar tu nota.

1. **Chat de Juego (5 décimas)**
   Dentro de las salas tiene que existir un chat al que todos los usuarios que esten dentro de la sala tienen acceso. Un usuario solo ve los mensajes que fueron enviados mientras estaba en la sala, es decir que no se deben mostrar los mensajes previos a la conexión del usuario en la sala.

   **Emojis** Su chat debe soportar el uso de 10 emojis especificados en la siguiente tabla:

| Imagen | Comando | Notas |
|---|---|---|
|  | :poop: | Todo con minúscula |
|  | O:) | una “O” en mayúscula |
|  | :D | Una “D” en mayúscula |
|  | ;) | |
|  | 8) | |
|  | U.U | Dos “U” en mayúsculas |
|  | :( | |
|  | 3:) | |
|  | o.o | Dos “o” en minúsculas |
|  | :v | Una “v” en minúscula |

   El sistema de *emojis* en su chat reconoce automáticamente los comandos de la tabla y al momento de enviar el mensaje, estos los cambia por el *emoji* adecuado, la única forma de evitar que el chat cambie algún comando por la imagen es insertar el comando dentro de los símbolos <> los cuales serán eliminados después de mandar el mensaje




2. **Desafíos (1 punto)**
   Un jugador podrá desafiar a otros usuarios que estén registrados en el sistema. El jugador que propone el desafío (el jugador desafiante) debe elegir una sala y un jugador para jugar una partida de 5 canciones. Cada jugador debe jugar la partida de 5 canciones de manera independiente (sin conocer lo que realizó el otro jugador) y luego se comparan sus puntajes obtenidos. Estos se calculan con la misma fórmula que para el modo normal.

   Luego de jugar de que ambos jugadores hayan terminado, se debe mostrar en una pantalla el puntaje para cada canción y el puntaje total para cada jugador. También deben mostrar el ganador de la partida. En caso de que el jugador desafiado no acepte el desafío, no se registrarán los puntajes y se le informará al jugador desafiante. Por simplicidad, guarde el puntaje como si cada combinación jugador-sala fuese una sala.

   Deberá agregar una nueva sección a la pantalla principal llamada desafíos, en la que aparezcan los desafíos que le hayan llegado al jugador y un pequeño menú para desafiar a otros jugadores.

# 7. Restricciones y alcances

- Tu programa debe ser desarrollado en Python 3.5

- Esta tarea es estrictamente individual, y está regida por el Código de Honor de la Escuela: Clickear para Leer.
- Su código debe seguir la guía de estilos `PEP8`.
- Si no se encuentra especificado en el enunciado, asuma que el uso de cualquier librería Python está prohibido. Pregunte por foro si se pueden usar librerías específicas.
- El ayudante puede castigar el puntaje[4] de su tarea, si le parece adecuado. Se recomienda ordenar el código y ser lo más claro y eficiente posible en la creación algoritmos.
- Debe adjuntar un archivo `README.md` donde comente sus alcances y el funcionamiento de su sistema (*i.e.* manual de usuario) de forma *concisa* y *clara*.
- Cree un módulo para cada conjunto de clases. Divídalas por las relaciones y los tipos que poseen en común. **Tareas que implementen el servidor o cliente en un sólo archivo serán castigadas fuertemente**.
- Cualquier aspecto no especificado queda a su criterio, siempre que no pase por encima de ningún otro.

# 8. Entrega

- **Fecha/hora:** 19 de Junio - 23:59 hrs
- **Lugar:** GIT - Carpeta: Tareas/T06

Tareas que no cumplan con las restricciones señaladas en este enunciado tendrán la calificación mínima (1.0).

[4] Hasta −5 décimas.