Explicaremos en cada apartado cada archivo de la práctica, excepto si se trata de un juego, donde se comentará cómo se juega y cómo ha sido programado:

Notificaciones.py

Lo empleamos para mostrar notificaciones cuando algún cliente envía un mensaje en el chat o para indicar eventos que ocurren.

No es posible pasar la clase *Notificaciones* a la clase del chat porque la librería no es *"pickable"* y, por tanto, no puede ser compartida entre procesos. Por este mismo motivo su implementación es interesante en esta práctica: si bien no crea procesos ni establece conexiones entre servidor y cliente, obliga a crear un túnel de comunicación entre el chat y el proceso inicial, para poder saber desde segundo los mensajes que recibe un cliente. Creemos que saber utilizar túneles y colas para transferir datos entre procesos complementa muy bien los conocimientos que tenemos sobre procesos y concurrencia.

Esta clase recibe dos parámetros: uno de ellos indica si el proceso que lo ejecuta es el cliente o el servidor y el segundo indica el sistema operativo. Este último parámetro era necesario porque no existe una librería universal para mostrar notificaciones en Linux y Windows, por lo que era necesario importar un paquete u otro según el sistema operativo:

1. **ToastNotifier:** paquete de notificaciones para Windows
2. **Notify2:** paquete de notificaciones para Linux.

Figura 1 Notificaciones mostradas cuando se recibe un mensaje (a la izquierda) o cuando el cliente se conecta al servidor (a la derecha). Windows 10.

Audio_client.py

La hemos empleado para poder establecer comunicaciones por voz con los participantes. Para enviar y recibir el audio del micrófono hemos necesitado dos librerías:

1. **Threading:** para crear un thread que grabe el audio y lo envíe
2. **PyAudio:** para capturar el audio del micrófono y para reproducir sonido

Hemos tenido que usar t*hreading* en vez de *multiprocessing* porque el segundo no hemos conseguido hacer que funcionase para ejecutar métodos de una clase que compartiesen un objeto PyAudio.

De manera experimental vimos que, cuando se estaba grabando el audio del micrófono con una tasa de muestreo de 20 000 Hz y 1024 *frames per buffer*, se enviaban constantemente 2048 bytes. Por ello, decidimos que el mejor protocolo para este caso era estar recibiendo siempre *trozos* de 2048 bytes y reproduciéndolos.

El servidor únicamente se encarga de recibir 2048 bytes de cada conexión abierta y reenviarlos a todos las otras conexiones.

Video_client.py

Se emplea para capturar el vídeo de la webcam y reenviarlo a los demás clientes. Se han necesitado los siguientes paquetes:

1. **Client de multiprocessing.connection:** para establecer conexiones con el servidor
2. **Process de multiprocessing:** para crear procesos
3. **Cv2:** para poder capturar y reproducir el vídeo
4. **Zlib:** para comprimir y descomprimir fotogramas
5. **NumPy:** para pasar las imágenes a bytes y poder enviarlas

La clase video consta de tres métodos: *__init__*, *envio_cam* y *recibir_cams*. La primera simplemente inicia el proceso para enviar la cámara y llama al método para recibir las cámaras de otros clientes. Como las imágenes que se graban pueden ser muy variadas, esta vez no podías contar con que el peso de cada imagen fuese constante. Por ello, hemos decidido emplear el protocolo siguiente para enviar cada fotograma:

1. El cliente graba un fotograma
2. Comprime el fotograma mediante el estándar JPEG (hemos optado por sacrificar mucha calidad de imagen porque hemos tenido muchos problemas con el ancho de banda)
3. Mediante el método *tobytes* de *numpy* convierte la imagen en una secuencia de bytes
4. Vuelve a comprimir el resultado mediante el algoritmo *zlib* (en nuestras pruebas, vimos que se conseguía reducir un 10% el tamaño de la imagen con este método)
5. Calcula cuánto ocupa el fotograma comprimido y envía, en una cadena con una longitud fija de 5 bytes, dicho tamaño al resto de clientes. Así, los clientes sabrán cuántos bytes tienen que recibir para completar 1 fotograma.
6. Envía 1 byte notificando su id al resto de clientes (para poder distinguir las imágenes que llegan de distintos clientes).
7. Finalmente, envía los bytes correspondientes al fotograma.

El método *recibir_cams* realiza el proceso inverso. Esto es:

1. Pide recibir 5 bytes y los convierte a tipo *int* para saber cuántos bytes ocupa el fotograma. Vamos a llamar al resultado *longitud*.
2. Pide recibir 1 byte para saber a quién corresponde el fotograma.
3. Pide recibir *longitud* bytes para recibir el fotograma completo.
4. Descomprime el fotograma
5. Mediante el método *frombuffer* convierte el fotograma de bytes a tipo *np.array*.
6. Muestra el fotograma

Como se están recibiendo y enviando constantemente fotogramas, era posible que ocurriera lo siguiente:

1. Clientes 2 y 3 envían la longitud de su fotograma, su id y su fotograma.
2. Servidor recibe los datos y se dispone a reenviarlos a Cliente 1:
    a. Servidor envía la longitud del fotograma de Cliente 2
    b. Servidor envía la longitud del fotograma de Cliente 3

Cuando el Cliente 1 recibiese la longitud del fotograma del Cliente 2, estaría esperando recibir su id (1 byte) y, sin embargo, recibe la longitud del fotograma del Cliente 3, por lo que se bloquea y devuelve un error. Para solventar este problema, en la lista de conexiones que crea

el servidor hemos añadido un semáforo para cada conexión del servidor. Así, cuando el servidor reenvía un fotograma a una conexión, adquiere el semáforo para la conexión, reenvía los 3 datos (tamaño, id y fotograma) y libera el semáforo, por lo que nos aseguramos de que se envían secuencialmente.

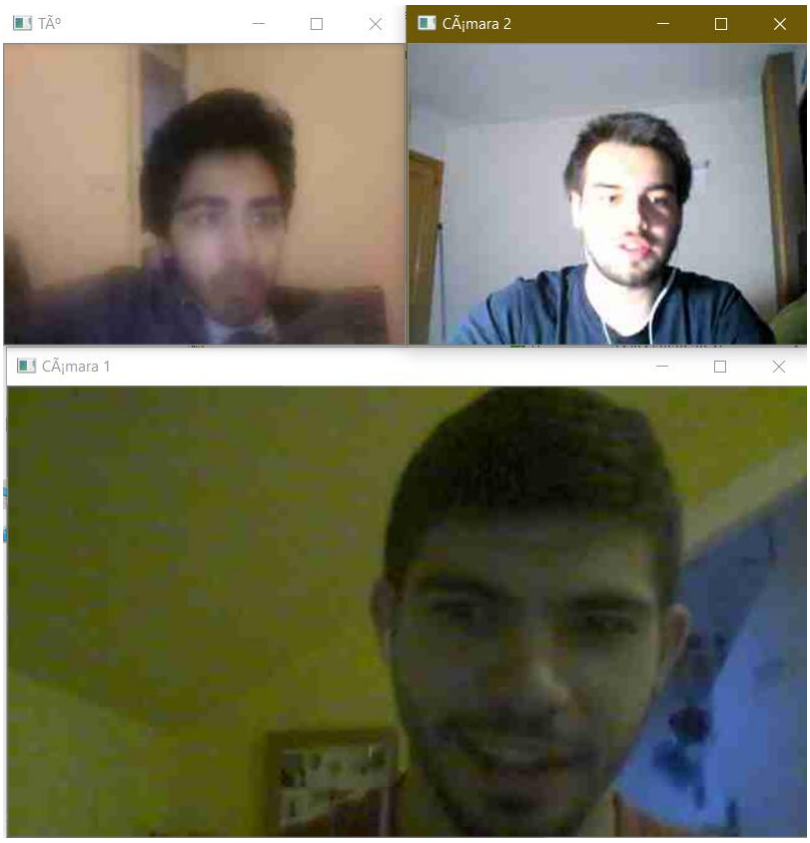


Figura 2 Transmisión de vídeo entre 3 clientes

Pizarra_client.py

Incluye la clase Minijuegos, que crea una ventana donde el cliente puede realizar varias acciones:

1. Dibujar en una pizarra compartida (con distintos colores, grosores, fondos…)
2. Hablar mediante el chat
3. Iniciar juegos: bomberman, pong, pong offline y rompeladrillos

Cuando se inicia se abre una ventana como la que sigue:

Figura 3. A la izquierda la pizarra. En el centro el chat. A la derecha un control para controlar el grosor del pincel. En la parte superior hay un menú donde seleccionar el color de fondo o pincel y para jugar a juegos.

Librerías empleadas:

1. **Client de multiprocessing.connection:** para establecer conexiones cuando el cliente quiere jugar a algún juego.
2. **Thread y Timer de threading:** para crear un thread que actualice la pizarra y el chat y para iniciar procesos retardados (sincronizamos así el inicio del juego Rompeladrillos).
3. **Pickle:** para convertir a bytes tuplas que contienen los comandos de la pizarra o mensajes del chat, entre otros.
4. **Tkinter:** para crear la ventana.
5. **Datetime:** para hacer operaciones con el tiempo. Se usa para sincronizar el inicio de Rompeladrillos.
6. **Tablero del archivo tablero.py:** para poder jugar al Bomberman.
7. **Pong del archivo pong.py:** para poder jugar al Pong.
8. **Rompeladrillos de rompeladrillos:** para jugar al Rompeladrillos.

Como en el archivo *audio_client.py*, aquí también nos hemos visto obligados a emplear la clase *Thread* en lugar de *Process* porque este último no permitía compartir en los parámetros los distintos widgets de *Tkinter*.

Cuando se inicia la clase *Minijuegos* se configura la ventana y se inicia un *Thread* que ejecuta el método *act_canvas*, que se encarga de recibir los comandos de otros clientes y aplicarlos en la pizarra compartida o el chat. Para poder distinguir los distintos comandos (los mensajes de chat, las líneas de la pizarra, los cambios de color del fondo…) hemos empleado el siguiente protocolo: cuando el cliente realiza cualquier operación (excepto jugar al rompeladrillos), envía primero el tamaño del comando y luego una n-upla cuya primera coordenada indica qué está haciendo el cliente:

- Si la primera coordenada es un 0, ha escrito un mensaje en el chat
- Si la primera coordenada es un 1, entonces el cliente está pintando en la pizarra.
- Si es un 2, entonces ha cambiado el color de fondo de la pizarra
- Si es un 3, ha borrado toda la pizarra.

Analicemos cada caso por separado:

1. **Caso 0: ha escrito un mensaje en el chat o ha mandado un emoji**

Debemos comprobar que el cliente no ha escrito un mensaje demasiado largo. En particular, no puede tener más de 10^{24} caracteres, pues de lo contrario la longitud del mensaje no cabría en 10 bytes y rompería el protocolo (detallaremos más adelante cómo funciona). En este caso, la n-upla que se envía es una 4-upla donde las coordenadas indican:

i. El tipo de acción (4 en este caso)
ii. El mensaje
iii. El nombre del usuario que envía el mensaje
iv. El ID del usuario que envía el mensaje

Nota: en Linux no hemos conseguido mostrar emoticonos. Es un problema con las fuentes que se utilizan.

2. **Caso 1: el cliente está pintando en la pizarra**
   En este caso la n-upla es una 7-upla, donde cada coordenada indica:
   i. El tipo de acción (en este caso es un 1)
   ii. La coordenada inicial X del ratón
   iii. La coordenada inicial Y del ratón
   iv. La coordenada final X del ratón
   v. La coordenada final Y del ratón
   vi. Grosor del pincel
   vii. Color del pincel
3. **Caso 2: ha cambiado el color de fondo**
   La n-upla es una tupla donde la segunda coordenada es el nuevo color de fondo.
4. **Caso 3: ha borrado toda la pizarra**
   Análogo, pero la segunda coordenada es el string *"Delete All"*.

El protocolo para enviar/recibir comandos es muy parecido al del vídeo, tanto que también requiere un semáforo en el lado del servidor para evitar el problema comentado antes. Cuando el cliente realiza alguna acción, envía primero 10 bytes indicando cuánto ocupa la n-upla que acabamos de describir y luego envía dicha n-upla convertida a bytes mediante la librería *pickle*. Cuando otro cliente recibe dicha n-upla, analiza los distintos casos y realiza la función que corresponda y añadiendo la n-upla a la cola de notificaciones si se trata de un mensaje de chat.

Sin embargo, si el cliente quiere jugar al rompeladrillos, le indica al servidor que la longitud del mensaje es 0 y al recibirlo el servidor se encarga de gestionar una partida de Rompeladrillos.

Cuando el cliente decide que quiere jugar a algún juego, se llevan a cabo otro tipo de operaciones, según de qué juego se trate. Supongamos que el cliente quiere jugar al bomberman, para lo cual accede al menú *Juegos* y hace click en el botón *Jugar al bomberman*.

Figura 4 Ventana al seleccionar 'Jugar al bomberman'. Obsérvense el mensaje del servidor en el chat y el nuevo botón que aparece a la derecha

El juego de bomberman admite *n* jugadores para cualquier número natural *n*, la única limitación es el servidor. Cuando selecciona la opción de jugar al bomberman, el servidor notifica a los demás clientes de que pueden unirse a la partida se lo desean. Cuando todos los clientes que quieran jugar se hayan conectado (puede jugar 1 persona sola si lo desea), cada uno debe hacer click en el nuevo botón que aparece (véase la figura 4) para indicar que está listo para jugar. Cuando todos los clientes están listos y pulsan el botón, se abre la siguiente ventana y se notifica a los demás clientes los nombres de los jugadores que van a empezar la partida:

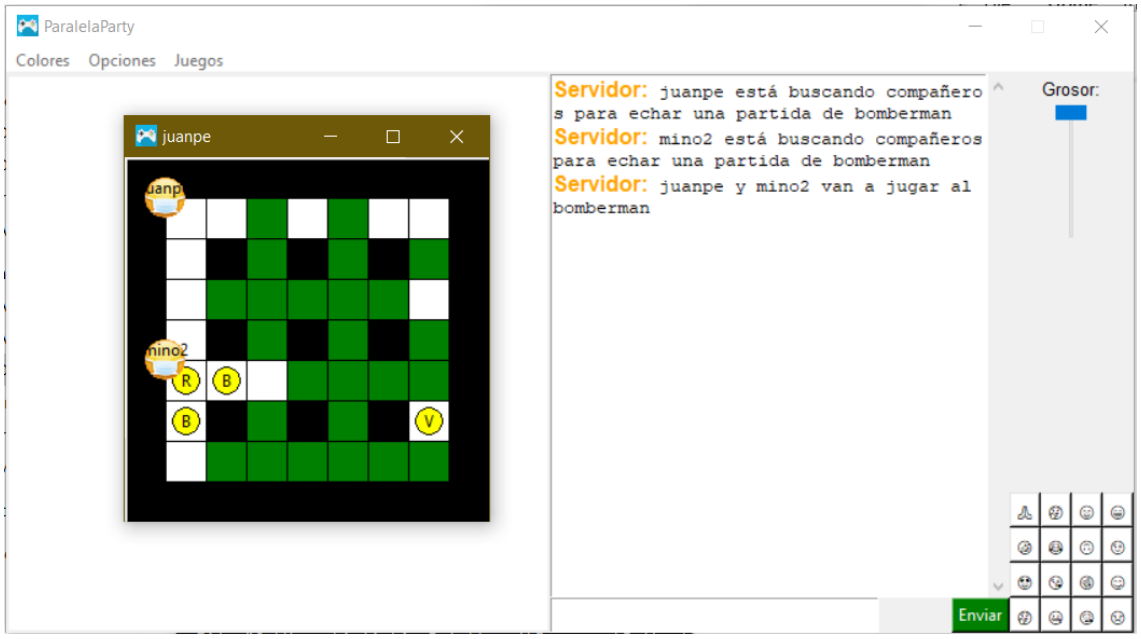


Figura 5 Inicio de una partida de bomberman entre dos clientes: mino2 y juanpe

Cuando el cliente hace click en *Jugar al bomberman* lo que ocurre internamente es lo siguiente:

1. El cliente se intenta conectar al servidor mediante el puerto 8488, que se ha reservado exclusivamente para el bomberman.
2. Cuando se conecta, envía su nombre al servidor.
3. Crea el botón verde '*Listo para jugar*', que cuando es presionado envía un mensaje al servidor indicando que el cliente está listo y borra el botón
4. Bloquea el proceso mientras espera a recibir los distintos datos para iniciar el juego (no los recibe hasta que todos los jugadores no estén listos): número de jugadores, tablero inicial, nombre de los jugadores… Aquí empleamos la función *recv* en vez de *recv_bytes* por simplicidad, porque sólo se envía un mensaje y no puede haber confusión entre el cliente y servidor.
5. Cuando recibe los datos, inicia el bomberman.

Lo mismo ocurre con el Pong, con la peculiaridad de que éste permite máximo dos jugadores por sesión. Hemos implementado el juego de tal manera que pueda haber varias sesiones de Pong simultáneas entre distintos jugadores, como se muestra en la figura 6.

Figura 6 2 partidas simultáneas de Pong. A la derecha, las notificaciones del servidor.

El límite de 25 caracteres en el nombre de los ususarios no es aleatorio: cuando un cliente se dispone a jugar al pong, envía 35 bytes con su nombre y recibe 35 bytes que contienen el nombre de su oponente. Según nuestras pruebas, 35 bytes equivalen a 25 caracteres cuando se emplea la librería *pickle* para serializar las cadenas de caracteres.

El rompeladrillos sigue una estructura distinta: este juego es solitario, pero los clientes pueden elegir ver la partida que juega un cliente. Para ello, era necesario coordinar que se iniciara el rompeladrillos exactamente al mismo tiempo en todos los ordenadores, porque MQTT solo permite guardar un mensaje (mediante el flag *retain*) y si un cliente se conectaba tarde no iba

a recibir el inicio de la partida ya empezada. Se han empleado para este fin las librerías *datetime* y *Timer*. Cuando un cliente decide jugar al Rompeladrillos ocurre lo siguiente:

1. Envía al servidor un *0*, indicando que quiere jugar al rompeladrillos
2. Espera a recibir una respuesta del servidor, que será una terna, siendo la primera coordenada un *4* (para distinguirlo de otros comandos), la segunda coordenada un *0* ó un *1* (si el cliente que lo recibe es jugador o espectador, respectivamente) y la última coordenada la hora a la que va a empezar la partida.
3. Indica en el chat que va a empezar una partida
4. Inicia un temporizador para iniciar la partida de rompeladrillos a la hora que establece el servidor

A los demás clientes les aparecerá un botón como el de la siguiente figura. Al hacerle click se borra el botón y se inicia la partida de rompeladrillos a la hora que establece el servidor.

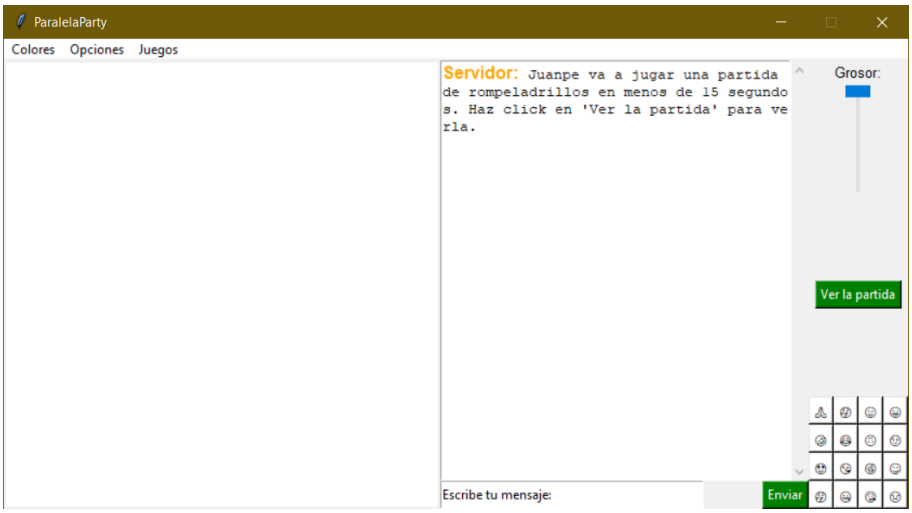


Figura 7 Botón para ver la partida de Rompeladrillos

Servidor.py

Es el archivo que debe ejecutar el servidor. Al abrirlo, es necesario que el usuario introduzca el sistema operativo que emplea para poder mostrar notificaciones. Librerías empleadas:

1. **Listener y AuthenticationError de multiprocessing.connection:** para poder establecer conexiones con los clientes y manejar errores cuando introducen una contraseña incorrecta.
2. **Process y Manager de multiprocessing:** para crear procesos que reciban datos de los clientes y crear listas compartidas entre los procesos.
3. **Sleep de time:** para *dormir* a un proceso cuando se ha alcanzado el número máximo de jugadores
4. **Datetime y timedelta de datetime:** para sincronizar el inicio de las partidas de Rompeladrillos. Timedelta se emplea para añadir un retraso a una hora dada.
5. **Notificaciones de notificaciones.py:** para mostrar notificacions sobre el estado del servidor.
6. **Crear_entorno_bomberman de bomberman_servidor.py:** para manejar conexiones de bomberman

7. **Crear_entorno_pong de pong_servidor.py:** para manejar conexiones de pong

Al iniciar el servidor se crean cinco Listeners distintos, todos compartiendo la misma IP pero con distinto puerto, desde el 8485 hasta el 8489. Estos listeners se emplean para el vídeo, audio, pizarra y chat (en adelante, pzch), el juego del bomberman y el juego del pong.

Se permiten un máximo de 8 jugadores. Cuando se alcanza esta cifra, a los nuevos jugadores se les mantiene en una sala de espera hasta que quede un hueco libre. Como necesitamos reenviar a todas las conexiones los datos que reciba el servidor de vídeo, audio, etc., necesitamos guardar en una lista compartida todas las distintas conexiones que se acepten. Además, para cada una de estas listas necesitamos un semáforo, pues si algún cliente se desconecta necesitamos borrar esa conexión de la lista y mientras permitir que se sumen otras conexiones simultáneamente.

Cuando se inicia el servidor se crean dos procesos: *crear_entorno_bomberman* y *crear_entorno_pong*, que se encargan de aceptar y manejar conexiones de clientes que quieran jugar a esos juegos. Mientras, el servidor acepta conexiones de vídeo, audio y pzch. Cuando se aceptan estas conexiones, se ejecutan tres procesos que se encargan de reenviar los datos de vídeo, audio y pzch a todos los clientes (excepto al emisor). En cada una de estas funciones, hemos manejado los errores mediante las funciones *try…except*. Nos hemos decantado por no especificar el tipo de error en el bloque *except* porque varía de sistema operativo en sistema operativo y según nuestra experiencia no es del todo fiable. Cuando se desconecta un cliente, se cierran todas las conexiones de vídeo, audio y pzch asociadas y se elimina dicho cliente de la lista de conexiones, siempre empleando semáforos para evitar escrituras simultáneas.

Merece especial mención la función *reenvio_pzch*, pues maneja las conexiones de rompeladrillos. Cuando recibe una longitud de mensaje nula manda a todos los clientes un mensaje indicando que uno de los jugadores va a empezar una partida de rompeladrillos. Le envía a todos los jugadores una terna, cuyas coordenadas son:

1. Un *4*, por los motivos que ya han sido explicados.
2. Un *0* ó *1* dependiendo de si se trata del cliente que ha enviado el mensaje o no, respectivamente.
3. La hora de inicio del juego. Para calcularla, el servidor toma su hora local y le añade 15 segundos mediante la función *timedelta*.

En cualquier otro caso, el servidor simplemente recibe los datos de pzch y los reenvía, siempre empleando semáforos para cerciorarse de que se envían todos los mensajes secuencialmente.

Bomberman_servidor.py

Se encarga de manejar conexiones de bomberman. Para ello, cuando se ejecuta crea un proceso que se encarga de aceptar conexiones de bomberman, mientras que el proceso principal espera a que todos los jugadores estén listos. Librerías empleadas:

1. **Random:** para coger posiciones aleatorias donde situar a los personajes en el tablero
2. **Pickle:** para convertir los bytes que recibe el servidor con el nombre de los jugadores a tipo string.
3. **Process, Manager y Queue de multiprocessing:** para iniciar procesos y crear listas compartidas entre ellos. La finalidad de Queue se explica a continuación.

Hemos añadido una cola a la que se le añade el primer jugador que diga que está listo porque el método *cola.get(True)* bloquea el proceso hasta que haya algún elemento dentro, y queríamos evitar el servidor estuviese constantemente ejecutando el siguiente bucle:

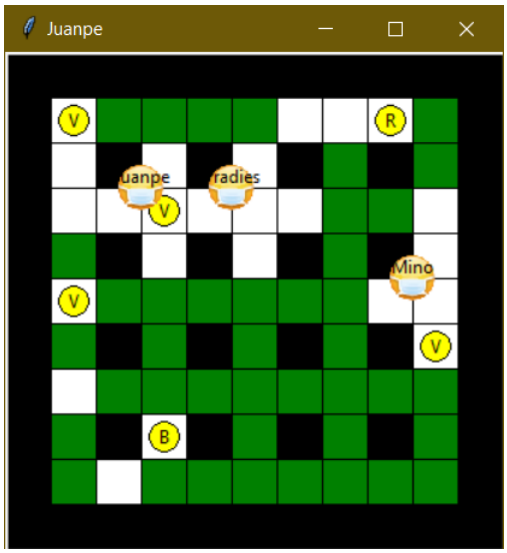
```
while True:
    if antesala:
        todos_listos = antesala[0][2]
```

Figura 8 Bucle en el proceso principal que comprueba si todos los jugadores están listos

porque sería un gasto innecesario de recursos. Cuando se conecta un cliente, lo siguiente ocurre en el servidor:

1. El servidor acepta la conexión y la añade a una lista que contiene todas las conexione
2. El servidor recibe el nombre del cliente mediante el método *recv_bytes*. El nombre se envía en 35 bytes (de ahí la limitación de caracteres de la que hablábamos antes) y era necesario emplear este protocolo porque el servidor se confundía cuando el cliente se conectaba y rápidamente pulsaba que estaba listo para jugar (se mezclaban el nombre y el byte que indica que el cliente está listo).
3. El servidor añade a una lista una terna con las siguientes coordenadas:
    i. Nombre del jugador
    ii. Número de jugador (la ID del jugador)
    iii. Booleano que indica si el jugador está listo
4. El servidor espera a recibir 1 byte del cliente, que le indicará que está listo.
5. Cuando el servidor recibe dicho byte, cambia la última coordenada de la terna anterior a *True* y añade al jugador (si es el primero en conectarse) a una cola para hacer que el servidor empiece a ejecutar el bucle de la figura 8, que comprueba constantemente si todos los jugadores están listos.
6. Cuando todos los jugadores están listos, envía los datos del tablero a los clientes y empieza un proceso para recibir y reenviar datos de los clientes.

Juego bomberman



Cómo jugar

1. Cada cliente debe poner su nombre cuando se le pida
2. Cuando el servidor lo requiera debes escribir la palabra: listo. Sin espacios ni comillas. En caso contrario dejaras de contar como jugador. La partida empezara cuando todos los jugadores estén listos, así que espera a tus amigos.
3. El objetivo es plantar bombas para matar a tus rivales, cuando eres el último hombre en pie has ganado la partida. Las mejoras que aparecerán en amarillo te ayudaran en tu misión, están en burbujas amarillas así que ¡ve a por ellas!
4. Los bloques verdes se pueden romper cuando una bomba los explota, los bloques negros no, así que mantente alejado de la explosión de las bombas de tus rivales y... ¡de las tuyas también! Estas explotaran en una cruz, si coges la mejora de radio, explotaran aún más lejos.

Como está programado el juego

Hay 3 clases diferentes:

1. Bomba
2. Personaje
3. Tablero

Estas clases emplean las siguientes librerías:

1. **Timer de threading:** para iniciar un contador al poner una bomba y que esta explote al terminar dicho temporizador.
2. **Thread de threading:** para poder actualizer el tablero cuando los clientes se mueven (*Process* no permite pasar este tipo de objetos como parámetro).
3. **Floor de math:** para hallar la posición más cercana a la posición del jugador
4. **Tkinter:** para crear la ventana con la interdaz del juego

1. **La clase *Bomba***

   Una bomba tiene 2 acciones diferentes: poner y explotar.

   Cuando un personaje pone una bomba esta se añade a la lista de bombas que hay activas en el mapa. Tras tres segundos la bomba explotará llamando a la función explotar.

   Cuando una bomba explota afecta en cruz a todo lo que esté en el rango de su explosión. Si la explosión alcanza una casilla negra, esta no se rompe y no deja atravesar la explosión. Si la explosión alcanza una casilla verde, rompe dicha casilla, pero no deja atravesar la explosión. Si la explosión alcanza una mejora esta desaparecerá del mapa y si alcanza a un jugador, este morirá.

2. **La clase *Personaje***

   Un personaje tiene 3 acciones diferentes: coger modificadores: (*get_bomb*, *get_range* y *get_speed*), plantar y moverse (*up*, *down*, *left* y *right*).

Cuando un jugador pasa por encima de un modificador lo coge y aumenta sus características (poder poner más bombas a la vez, tener más rango en la explosión de sus bombas o poder moverse más rápido por el mapa).

Cuando el jugador ejecuta *plantar*, pone debajo de sus pies una bomba siempre y cuando esté vivo y si no haya llegado al máximo de bombas activas en ese momento.

El jugador podrá moverse por el mapa teniendo en cuenta que no puede atravesar casillas negras ni verdes. Puede moverse por encima de la bomba que acaba de poner, pero una vez sale de encima de esta, ya no podrá volver a atravesar ninguna bomba más (así que ten cuidado y no te quedes encerrado por las bombas o morirás). Cuando un jugador pasa por encima de una mejora, la coge (ver apartado coger modificadores). Es posible atravesar a otros jugadores.

3. La clase *Tablero*

Un tablero tiene seis métodos diferentes: recibir y actualizar, crear personaje, dibujar tablero, dibujar jugadores, mover personaje ajeno y mover:

- **Recibir y actualizar:** mediante la función *recibir_y_act*, se recibe el par (*player*:Int, *action*:Int), y además hace que el jugador *player* realice la acción action.

- **Crear personaje:** el tablero crea un objeto de la clase personaje con el número, nombre y posición que se le diga. Estos se generan por el servidor y sólo el nombre es elegido por el usuario. Cada personaje se crea en una posición inicial (previamente preparada para que siempre tenga una opción de plantar y no ser explotado por su propia bomba).

- **Dibujar tablero:** cada posición de la matriz se corresponde con un cuadrado de 30x30 píxeles. El tablero es aleatorio y su tamaño depende del número de jugadores. Cada partida es totalmente diferente. El dibujo a partir de la matriz tablero es el siguiente:

    - 0: se dibuja una casilla en blanco
    - -2: se dibuja una casilla en verde
    - -3: se dibuja una casilla negra
    - n>0: se dibuja un coronavirus. ¡Huye de él! ¡son las bombas! La bomba i-ésima corresponde al jugador i-ésimo
    - b: se dibuja la mejora de bomba
    - r: se dibuja la mejora de radio
    - v: se dibuja la mejora de velocidad

  Nota: puedes saber qué jugador eres mirando en la ventana que se abre automáticamente. Ahí encontrarás tu nombre.

- **Dibujar jugadores:** esta función fue creada porque dibujar el tablero en cada movimiento de cada jugador es muy poco eficiente. Por ello hay una primera lista que almacena todo lo creado en el proceso de dibujar a los jugadores. Borra todo lo creado (sobre los jugadores) y vuelve a dibujar únicamente los jugadores,

almacenando cada objeto en dicha lista para poder ser borrado posteriormente y así ahorrarnos borrar todo el tablero.

- **Mover personaje ajeno:** esta función va de la mano de la de *recibir y actualizar*. Dado un personaje y una acción, dicho personaje realiza la acción, permitiendo así mover a tiempo real a los jugadores que no sean el propio.
- **Mover:** esta función permite mover nuestro personaje (i-ésimo) por el mapa utilizando: a, *w*, *s* y *d* ó los cursores del teclado, según prefiera el cliente. Se planta una bomba pulsando la tecla espacio. Tras realizar una acción, se le manda al servidor el par (*acción*, *i*) (cada número corresponde a una acción (*)) y este la reenvía todos los jugadores (excepto al emisor) para que puedan, mediante la función *mover ajeno*, mover al jugador que ha enviado el comando.
  (*) El código de cada acción es el siguiente:
  1. Moverse hacia arriba
  2. Moverse hacia abajo
  3. Moverse hacia la izquierda
  4. Moverse hacia la derecha
  5. Plantar bomba

Pong_servidor.py

Se encarga de crear un entorno para jugar al pong dentro de *pizarra_client.py*, además de reenviar los comandos que ejecuta cada cliente durante el juego.

Este fichero importa las siguientes librerías:

1. **Listener de multiprocessing.connection:** para establecer conexiones con cada cliente.
2. **Process de multiprocessing:** para crear procesos
3. **Manager de multiprocessing:** para crear recursos compartidos que puedan modificar todos los clientes.
4. **AuthenticationError de multiprocessing:** para evitar errores de autentificación cuando el servidor se conecte con el cliente.

Este fichero se compone de tres funciones que cada una lleva a cabo una tarea para dotar al juego de una faceta multijugador. Las detallamos a continuación:

1. **reenvio_pong:** tiene como argumentos la conexión de los dos clientes que inician el juego. Mientras los dos clientes se encuentren conectados, esta función reenviará los comandos que envíe uno de los clientes a su oponente para dibujar los movimientos que realizan en su raqueta.

   Cuando uno de los dos clientes se desconecta, entra por la excepción. Entonces el booleano “enviando” se vuelve falso, se sale del bucle while y el servidor cierra su conexión.

2. **esperar_listo:** recibe como argumentos la conexión (del cliente que espere listo) y la lista con los nombres de los clientes. Cuando un cliente está listo para jugar al pong, se conecta desde fichero *pizarra_client.py*, en particular con la función *conectar_pong*. Al principio, *esperar_listo* envía los datos de la id de la conexión con el cliente a *conectar_pong*, y después desde cada conexión recibe el nombre del cliente y los escribe en una lista compartida de nombres.

3. **crear_entorno_pong:** recibe como argumento el listener, ya que esta función está importada en el servidor central de la plataforma, y actúa desde ahí. Esto es para que puedan procesarse varias partidas simultáneamente, con dos jugadores cada una.
   La función para crear entorno de partida se basa en un bucle While que siempre está corriendo. Primero creamos una lista que recibe las conexiones de los clientes que quieren comenzar la partida y una lista que le asigna las ids. Con estos dos parámetros creamos una lista de tuplas (id, conexión) de cada cliente que se conecta al entorno del juego. Generamos además una lista compartida para insertarle los nombres de cada cliente en la función *esperar_listo*.
   Después iniciamos un proceso *esperar_listo* por cada cliente que se conecta y los ejecuta.
   Finalmente, a cada jugador de la lista le asigna el id del oponente, le envía su nombre y empieza el proceso de *reenvio_pong*, para empezar a conectar cada cliente con este servidor.

Pong

Cómo jugar

El objetivo del juego es marcar más puntos que el rival. Marcas un punto cuando consigues que la bola llegue al lado contrario del que está tu pala sin que el otro jugador la haya podido parar, en ese caso ganas un punto.

Cómo está programado el juego (offline – pong_offline.py)

Para evitar el uso de variables globales, se crea la clase *Pong_offline*. Esta hace uso del módulo *turtle* de Python.

1. ***Objetos***

   Contiene una pantalla, a la que se le da un tamaño de 600 x 400 píxeles.

   La bola, las raquetas y el marcador son objetos *Turtle*. Para evitar que escriban al moverse se hace función de *penup*.

   El *tracer* es puesto a 0 para apagarlo, lo que nos será útil más adelante.

2. ***Movimiento e input***

   Para mover cada raqueta, se usan las teclas de arriba o abajo, o las letras “w” y “s” (dependiendo de la pala). Para esto se usan 4 funciones, donde cada una representa un movimiento (arriba o abajo) de cada una de las palas. Después, se usa listen para alertar a pantalla. Y, mediante onkeypress se ejecuta cualquiera de las 4 funciones de movimiento cuando se presione cualquiera de las letras.

3. ***Bucle principal del juego***
   El bucle principal del juego se encarga, primero, de actualizar el juego, de ahí el uso de la función update, y por qué hemos tenido que apagar el tracer. Después, se encarga de

mover la bola según la velocidad y posición actual (por ejemplo, rebotando si llega a los lados).

También se encarga de detectar colisiones. Debido a que las raquetas que hemos creado son objetos puntuales, pero queremos que tengan una altura de 60 píxeles, queremos ver si está a la altura correcta en el eje de las x y de las y (con sus correspondientes anchuras) para ver si hay colisión. En caso afirmativo rebota con una velocidad aumentada.

Finalmente, se encarga de actualizar el último objeto *turtle*: el marcador, en caso de que la bola pase de los límites del eje de las x (300 o -300 píxeles), borra lo anterior y luego escribe la nueva puntuación.

Cómo está programado el juego (multijugador – pong.py)

Este fichero importa las siguientes librerías:

1. **turtle:** para desarrollar la interfaz gráfica del juego.
2. **Queue de multiprocessing:** cola para almacenar los comandos que le reenvía el fichero *pong_servidor.*
3. **Empty de queue:** para definir una cola vacía.
4. **Thread threading:** para inicia un hilo que actualice las posiciones de las raquetas.

La función que inicia la clase recibe los siguientes argumentos:

1. **conn:** conexión que tiene cada Cliente que en *conectar_pong* del archivo cliente de la pizarra y chat.
2. **id:** La id que le asigna *conectar_pong* del archivo cliente de la pizarra y chat.
3. **mi_nombre:** nombre que se asigna cada cliente al entrar en la ventana inicial.
4. **nombre_oponente:** nombre del usuario que se asignó en la ventana inicial.

Al iniciarse esta clase, se definen primero la conexión, las id del usuario y del contrario y se establece la cola de comandos. Después se crean los distintos objetos mediante la librería *turtle*, (las dos raquetas, la bola y el marcador) y definimos los botones que activarán las funciones de subir y bajar las raquetas.

Iniciamos el hilo con la función *recibir_raquetas* para recibir los comandos que reenvía el *pong_servidor.* Esta función se encarga de tomar esos comandos en forma de bytes, en concreto 1 byte y de ponerlos en la cola.

Entonces, comienza el bucle principal del juego definiendo la posición de la bola dadas sus coordenadas de posición y velocidad en los ejes.

Si la cola de comandos no está vacía, tomaremos el primer elemento que estará en un formato de bytes, lo pasamos a entero. En función del entero que haya recibido, se ejecutará una función:

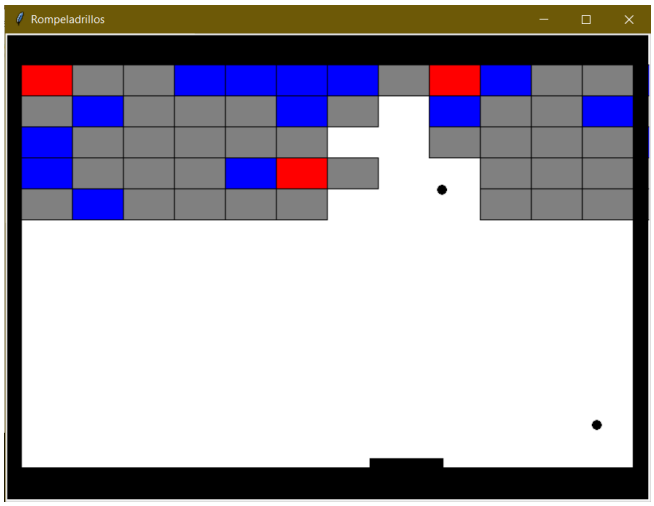
- **Si com == 0:** el cliente que envió el comando 0 subió la raqueta en su pantalla y por lo tanto debemos subir la raqueta en la pantalla del oponente.
- **Si com == 1:** el cliente que envió el comando 1 bajó la raqueta en su pantalla y por lo tanto debemos bajar la raqueta en la pantalla del oponente.

Definimos a continuación los rebotes de la bola cuando toca los límites superior e inferior la pantalla, cambiando de sentido en la componente vertical, y los goles cuando se sobrepasan los límites laterales de la pantalla cambiando la puntuación en el marcador.

El resto de las funciones están fuera del bucle. Las explicamos a continuación:

1. **Recibir_raqueta:** pone en la cola los comandos recibidos por el servidor
2. **Raqueta_arriba:** al presionar el cliente el botón *Up* (cursor hacia arriba) sube su raqueta una posición mientras esté dentro de los límites de la pantalla y envía esta información a *pong_servidor.*
3. **Raqueta_abajo:** funciona de manera análoga a la anterior, presionando el botón *Down* (cursor hacia abajo).
4. **Raqueta_arriba_Contr:** sube la raqueta del oponente una posición cuando ha recibido el comando pertinente.
5. **Raqueta_abajo_Contr:** funciona de manera análoga a la función anterior.

Rompeladrillos



Cómo jugar

Es un juego clásico dónde dispones de una barra que puedes mover con los cursores izquierda y derecha. El objetivo es no dejar caer la bola mientras vas rompiendo todos los bloques, en este caso para hacerlo algo más interesante cada cierto tiempo aparece una nueva bola para ayudarnos con nuestra tarea, perdemos si en algún momento nos quedamos sin bolas y ganamos cuando todos los bloques hayan sido destruidos. Los bloques tienen diferentes colores que muestran su estado, los bloques solo se destruirán cuando estén de color gris, si están de otro color les bajaremos en un nivel su estado y cambiarán de color, hay 5 estados, gris, azul, rojo, amarillo, cian y magenta.

Por qué ha sido implementado

Queríamos abarcar todas las áreas vistas en clase y con los juegos y funciones anteriores nos faltaba emplear el protocolo MQTT. Para aprender a manejarlo, decidimos implementar un juego solitario donde el jugador podía retransmitir su partida a través de MQTT que, si bien no

está destinado para este fin, creemos que es una manera creativa de implementarlo y encajaba con la temática del proyecto.

Cómo está programado el juego

Hay 3 clases diferentes:

- Bola de *bola_ladrillos.py*
- Barra de *barra_y_bloque_ladrillos.py*
- Bloque de *barra_y_bloque_ladrillos.py*

Estas clases emplean las siguientes librerías:

1. **Tk de tkinter:** para crear la ventana de juego.
2. **Canvas de tkinter:** crear el tablero.
3. **Random:** para generar bolas aleatorias y la velocidad de la bola.
4. **Time:** para el envejecimiento de los ladrillos y la aparición de las bolas.
5. **Bola de bola_ladrillos:** acceder a la clase de la pelota.
6. **Barra de bola_ladrillos:** acceder a la clase de la barra del juego.
7. **Bloque de bola_ladrillos:** acceder a la clase de los ladrillos.

Para la creación del programa hemos usado la librería *tkinter* que nos va a proporcionar el lienzo donde proyectar nuestro juego, además de varias funciones que usan el *multiprocessing* para mover los objetos dentro de este lienzo. El juego viene determinado por una matriz que nos dirá si en una posición hay bloque que se puede romper, uno que no se puede o simplemente no hay nada (esta matriz se genera con las primeras funciones del programa *zerolistmaker(n)* que nos crea una lista de ceros, *matriz_ceros()* que nos crea una lista de listas que será la matriz y *generar()* que cambiará algunos ceros por otros números que representarán los distintos bloques). A continuación, se crean tres clases que explicaremos por separado:

- **Clase Bola:** tiene 9 atributos, uno que será el lienzo, otro de velocidad, 4 de posición, 2 de dirección y el último que indicará el estado. A continuación, nombraremos y explicaremos brevemente las funciones implementadas dentro de la clase:
  - ***Chocar_bloque_izquierda, chocar_bloque_derecha, chocar_bloque_abajo, chocar_bloque_arriba***: estas funciones nos indican si un bloque cualquiera (que son las posiciones de la matriz con un 1) pueden ser impactados en uno de sus lados por la bola en un momento preciso sabiendo la posición de la bola, además le cambia la dirección a la bola y le baja un estado al bloque si de verdad se produce el impacto.
  - La función ***movimiento*** nos va a proporcionar, como su propio nombre indica, el desplazamiento de la bola. Utiliza la función ***move*** de la librería, la cual usa *multiprocessing* moviendo la bola en una dirección dada, usamos la función **after**, también presente en la librería para llamar a la función *movimiento* cada cierto tiempo, que es lo que llamaremos velocidad de la bola.
  - ***Posición_barra*** nos relaciona una función de la clase barra con esta clase para poder usarla dentro de esta.
  - ***Borra*** es una función auxiliar que nos busca cierto elemento en una lista.

- **Clase Barra:** esta clase representa la barra que nos ayudará a salvar la bola, tiene 4 atributos, el lienzo, la posición, el estado y la velocidad. A continuación veremos las funciones dentro de la clase:
    - ***Left*** y ***right*** son las funciones que se encargarán de mover la barra de izquierda a derecha. Emplean la función ***move***.
    - ***Mover*** relaciona las dos funciones anteriores con dos teclas del teclado, en este caso son los cursores de dirección izquierda y derecha del teclado.
- **Clase Bloque:** cada rectángulo de colores será un elemento de esta clase. Tiene 4 atributos, el lienzo, el estado y 2 posiciones. Esta clase no tiene métodos dentro de ella.

A continuación, se procede a crear la ventana que actuará como nuestro lienzo del tamaño que nosotros le proporcionemos y la función *dibujar*. Esta función nos pinta los bloques (rectángulos) según las posiciones y valores que tengan en la matriz (pueden ser -1, -2 o -3). Nos pintará rectángulos de color negro de diferentes tamaños según su valor, si el valor de la matriz es 1 generará un objeto *Bloque* que pintará automáticamente un rectángulo gris y se añadirá a una lista global, es importante ver que los bloques negros no pertenecen a la clase *Bloque*, mientras que los grises sí. La siguiente función es *borrar* que nos permite borrar un bloque gris del lienzo. Por otra parte, ***buff_bloque*** nos sube el estado en +1 de un *Bloque* aleatorio cada cierto tiempo, lo cual implica un cambio de color; ***debuff_bloque*** hace lo contrario, pero esta funciona como una función auxiliar que se usará a la hora del impacto de una Bola con un Bloque. ***Crear_bolas*** es la función encargada de generar cada cierto tiempo una nueva Bola y ponerla en movimiento y, por último, ***romper_ventana*** es la encargada de cerrar el juego cuando nos hemos quedado sin bolas, es decir, cuando perdemos. Con esto concluimos el programa.

Barra_y_bloque_ladrillos.py

Comentamos este archivo aparte para explicar la comunicación entre los clientes mediante MQTT. Las librerías que utilizamos en este archivo son las siguientes:

1. **Client de paho.mqtt.client:** necesario para utilizar la herramienta del bróker
2. **Thread de threading**: para inicia un hilo que actualice el tablero

La clase *Barra* se compone de las siguientes funciones:

1. En la función **__init__** de la clase barra definiremos la posición de la barra, la velocidad de la bola y el topic al que se suscriben el jugador y los espectadores para recibir comandos y proyectar la partida.
   Si el usuario que se suscribe al topic es espectador (hay un booleano que se toma como parámetro que es *True* si el cliente que lo ejecuta es espectador y *False* en caso contrario), solo podrá recibir comandos y visualizarlos en su pantalla, ejecutando un hilo con la función ***rec*** (recibir).
   En cambio, si es el jugador podrá controlar la partida con el teclado, pudiendo empezar la partida a través del hilo *empezar_client*.
2. **empezar_client:** esta función hace que cada usuario se conecte al bróker online *"broker.hivemq.com".*

3. **on_message**: función que recibe los comandos que le reenvía el broker. Si el comando es 0, moverá a la izquierda la barra del espectador y si recibe un 1, lo moverá a la derecha
4. **rec:** Esta función conecta a los espectadores al juego rompeladrillos. El espectador se suscribe al topic definido en el método *init* y recibe los comandos a través de la función *on_message*.

5. **left_espectador**: función usada para mover un objeto de la clase *Barra* un número de posiciones, que vienen determinados por la velocidad, a la izquierda cada vez que se ejecute esta función. También varía de igual manera la posición de la Barra.
6. **right_espectador:** misma función que la anterior, pero sirve para el desplazamiento hacia la derecha de la Barra.
7. **left:** función usada para mover un objeto de la clase Barra un número de posiciones, que vienen determinados por la velocidad, a la izquierda cada vez que se ejecute esta función. También varía de igual manera la posición de la Barra. Además, publica un mensaje con un 0 para indicar a los suscriptores que ha movido la barra.
8. **right:** misma función que la anterior, pero sirve para el desplazamiento hacia la derecha de la barra. En este caso publica un 1.
9. **mover:** relaciona las dos funciones anteriores con las teclas de dirección del teclado *right* y *left*.
10. **posición:** función auxiliar para llevar el atributo de posición a otra clase.

Cliente.py

Librerías importadas en el archivo:

1. **Client de multiprocessing.connection:** establecer conexiones con el servidor.
2. **Process de multiprocessing:** establecer procesos para audio, video, juegos, chat y pizarra.
3. **Queue de multiprocessing:** establecer una cola para las notificaciones.
4. **Manager de multiprocessing:** almacenar la información del cliente para poder usarla en el resto de los archivos de la plataforma.
5. **Audio de audio_client:** audio entre clientes.
6. **Video video_client:** vídeo entre clientes.
7. **Minijuegos de pizarra_client:** clase de la pizarra online.
8. **Notificaciones de notificaciones:** recibir notificaciones emergentes.
9. *** de tkinter:** interfaz gráfica de las ventanas.
10. **Messagebox de tkinter:** ventanas emergentes referentes a errores en el inicio del cliente.
11. **Font de tkinter:** fuente y temaño de la letra en la ventana de menú.
12. **Sys:** para cerrar el programa si el usuario ha cerrado la ventana inicial sin introducir los datos.

El cliente se inicia y se abre un menú de opciones donde el cliente deberá introducir el nombre o apodo que desee como usuario de la plataforma *ParalelaParty*, la dirección IP del servidor al que se quiere conectar y el sistema operativo utilizado en su ordenador.

Figura 9 Ventana inicial del proyecto

La elección del sistema operativo servirá para recibir notificaciones emergentes tanto para Windows como para Ubuntu. Si el usuario se conecta usando Windows, se importará la librería *win10toast*, especial para notificaciones emergentes, por lo que deberá tenerla instalada previamente el usuario en su ordenador. Si por el contrario el cliente se conecta usando Ubuntu se importará la librería *notify2*, que también deberá estar previamente instalada.

El cliente no podrá acceder a la plataforma si no rellena los tres datos que aparecen en el menú, y saltan ventanas emergentes con explicaciones específicas de cada error en función de los datos que se rellenen. Conseguimos que la plataforma no se inicie completamente antes de completar los datos del menú de selección, utilizado el método *join*, que garantiza que el proceso de la ventana inicial debe acabarse antes de comenzar el resto del programa. Además, guardaremos la información introducida por el cliente en una lista de *Manager*, con el fin de usarla para las notificaciones emergentes del cliente y el chat, la conexión al servidor y los nombres personalizados en juegos y chat.

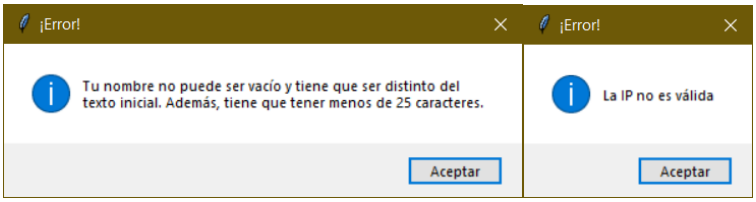


Figura 10 Algunas ventanas emergentes indicando errores

Una vez completados los aspectos anteriores, se iniciarán tres procesos nuevos distintos, uno por la conexión de video entre usuarios (puerto 8485), otro para el audio (puerto 8486) consiguiendo con estos dos la videollamada, y un tercero referente a la pizarra y chat (puerto 8487) a través de los cuales nos podremos conectar a los distintos juegos.

Finalmente, empleamos un bucle infinito para recibir todas las notificaciones emergentes referentes a los mensajes de otros usuarios en el chat. Este bucle no consume muchos recursos puesto que el proceso se bloquea cuando la cola de notificaciones está vacía.