IIC2233 • PROGRAMACIÓN AVANZADA

INTERFACES GRÁFICAS II & NETWORKING

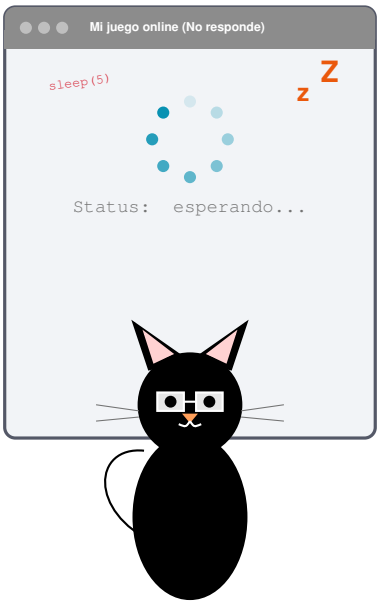
event loop • `QTimer` • `QThread` • `QMutex`

sockets + GUI • *front-end* y *back-end* • recetario

una ventana que nunca se congela, aunque el servidor tarde en responder

PyQt6 · socket · JSON





Felipín quiso conectar su juego a un servidor. Puso el `recv()` directo en el *slot* de un botón y la ventana se congeló. Lo movió a un *thread*... que modificaba los *labels* por su cuenta: `Segmentation fault`. Al final guardó el *thread* en una variable local y PyQt le gritó `QThread: Destroyed while thread is still running`.

No seas como Felipín. Aquí aprenderás cómo funciona el ***event loop***, la **regla de oro** de los *threads* en PyQt, cuándo usar **`QTimer`** o **`QThread`**, y cómo armar un **cliente con interfaz** que conversa con un servidor sin congelarse.

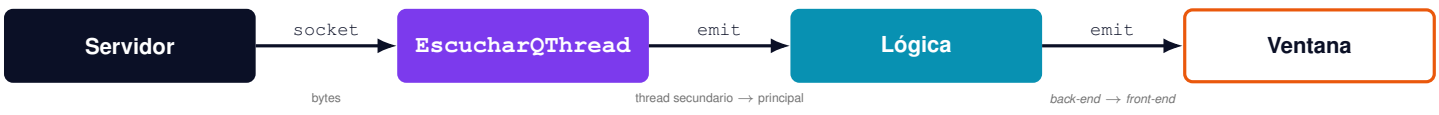
¿Qué herramienta uso?

Reacción rápida
→ un *slot* normal

Periódico y rápido
→ `QTimer`

Bloquea o tarda
→ `QThread` + señales

si escribes un `sleep` en un slot, algo anda mal





Resumen de Contenidos

IIC2233 — Programación Avanzada • Edición Señal en vivo

Resumen elaborado por el equipo de ayudantía, en base al material oficial del curso

Tabla de Contenidos

1 Introducción **3**

2 El *event loop* y la regla de oro **3**
2.1 Repaso: ¿qué hace `app.exec()`? . . . 4
2.2 Una GUI congelada . . . 4
2.3 ¿Y si usamos un *thread*? . . . 5
2.4 La regla de oro . . . 6
2.5 ¿Necesito un *thread*? . . . 8
2.6 *Debuggeo*: el *hook* de excepciones . . . 8

3 `QTimer`: lo periódico, sin *threads* **10**
3.1 ¿Qué es un `QTimer`? . . . 10
3.2 Reloj digital . . . 11
3.3 Múltiples `QTimer` . . . 11
3.4 Una sola vez: `singleShot` . . . 13
3.5 Pausar y reanudar . . . 13
3.6 *Loop* de juego con un solo `QTimer` . . . 14

4 `QThread`: lo que bloquea, con señales **17**
4.1 *Threads* y PyQt . . . 17
4.2 `QThread` . . . 18
4.3 Afinidad: ¿en qué *thread* se ejecuta cada método? . . . 19
4.4 Múltiples `QThread` y la referencia . . . 23
4.5 Detener un `QThread` y cerrar limpio . . . 24
4.6 `QMutex` . . . 25

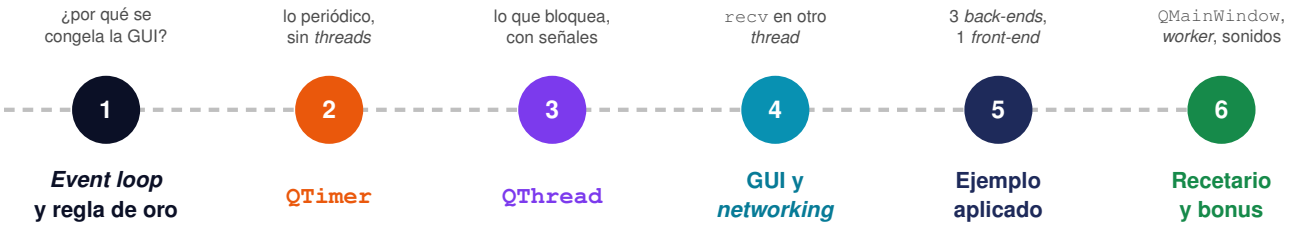
5 Interfaces gráficas y *networking* **27**
5.1 La arquitectura: ¿quién tiene interfaz? . . . 27
5.2 El protocolo . . . 28
5.3 Ejemplo simple . . . 28
5.4 Ejemplo avanzado: interacción entre clientes . . . 31

6 Ejemplo aplicado: un *front-end*, tres *back-ends* **34**
6.1 Comparando los tres *back-ends* . . . 35

7 Recetario: *tips* útiles **36**
7.1 `QMainWindow` . . . 36
7.2 `isAutoRepeat` en `keyPressEvent` . . . 38
7.3 Versiones de PyQt . . . 39

8 Bonus: patrón *worker* y sonidos **39**
8.1 El patrón *worker*: `moveToThread` . . . 39
8.2 Sonidos en PyQt6 . . . 40

9 Errores frecuentes y hoja de referencia **41**
9.1 Errores frecuentes: síntoma → causa → solución . . . 41
9.2 Hoja de referencia . . . 42



1 Introducción

La semana pasada construimos nuestras primeras ventanas: *widgets*, *layouts*, señales y la separación *front-end/back-end*. Todo funcionaba porque cada reacción era **instantánea**: un clic cambiaba un texto y listo. Pero apenas un programa tiene que **esperar** algo (un mensaje por la red, un cálculo pesado, "mover un personaje cada 50 ms"), la ventana empieza a congelarse, a caerse con errores rarísimos o a portarse distinto en cada ejecución.

Este resumen explica **por qué** pasa eso y **cómo** evitarlo. Todo se construye sobre una sola idea: el ***event loop*** de PyQt corre en **un único *thread***, y hay que cuidarlo. A partir de ahí aparecen las dos herramientas de la semana (`QTimer` y `QThread`) y, al final, la unión de todo con lo que ya sabes de *networking*: un **cliente con interfaz gráfica** que conversa con un servidor.

■ Los bloques de este resumen

- ***Event loop* y regla de oro** (Sección 2): por qué se congela una GUI, por qué un *thread* no puede tocar *widgets*, cómo decidir si necesitas un *thread*, y el *hook* de excepciones.
- **`QTimer`** (Sección 3): tareas periódicas sin *threads*: reloj, varios *timers*, `singleShot`, pausa y el *loop* de juego.
- **`QThread`** (Sección 4): *threads* que avisan con señales, **afinidad**, dónde crear los objetos, referencias, cierre limpio y `QMutex`.
- **GUI y *networking*** (Sección 5): protocolo, un *thread* que escucha, desconexiones, y un servidor que envía a todos los clientes.
- **Ejemplo aplicado** (Sección 6): el mismo juego con tres *back-ends* distintos.
- **Recetario, bonus y referencia** (Secciones 7 a 9): `QMainWindow`, `isAutoRepeat`, PyQt5 vs. PyQt6, patrón *worker*, sonidos, errores frecuentes y hoja de referencia.

★ Cómo leer este resumen: el código de colores

Los colores no son decoración, **significan algo** en todos los diagramas: **azul medianoche** = *event loop* y *thread* principal; **coral** = `QTimer`; **violeta** = *threads* secundarios; **cian** = señales y red. Además, los recuadros **rojos** marcan trampas, los **coral** son definiciones, los **violeta** son analogías, los **verdes** describen lo que verás al ejecutar, y los bloques negros con >_ muestran la salida de la terminal.

▲ Ejecuta los ejemplos en tu computador, no en Jupyter

Jupyter no fue creado para ejecutar interfaces de escritorio ni programas de red, así que todos los ejemplos se prueban como *scripts* `.py` desde tu terminal. La terminal debe estar **posicionada en la misma carpeta** del archivo (sobre todo si carga imágenes o sonidos con *paths* relativos). Los *scripts* están en la carpeta `scripts/` del material:

>_ Terminal

```
cd scripts/1-event-loop
python3 1_gui_congelada.py
```

2 El *event loop* y la regla de oro

2.1 Repaso: ¿qué hace `app.exec()`?

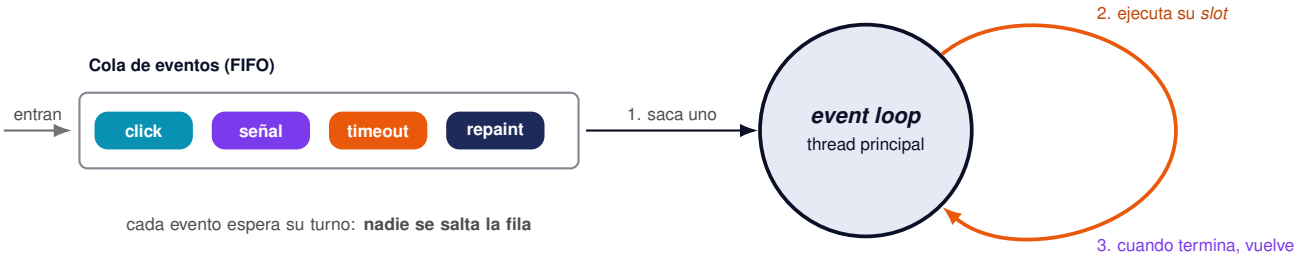
La semana pasada vimos que `app.exec()` inicia el ***main loop*** o ***event loop*** de PyQt. Ahora toca entender bien qué hace, porque de eso depende todo lo demás.

El *event loop*

Es un ciclo que, mientras la aplicación esté abierta, repite:

1. Saca **un** evento de la **cola de eventos**: un clic, una tecla, una señal emitida, la orden de redibujar la ventana, etc.
2. Ejecuta el método (*slot*) conectado a ese evento.
3. Cuando ese método **termina**, vuelve al paso 1.

Todo esto ocurre en **un único *thread***: el ***thread* principal**, el que ejecutó `app.exec()`.



Esto significa que **todos** los métodos conectados a señales de la interfaz, todos los `mousePressEvent` y `keyPressEvent`, e incluso el **dibujo** de la ventana, se ejecutan **uno después de otro** en ese mismo *thread*.

☞ Analogía: la cajera única

El *event loop* es un supermercado con **una sola caja**. Cada evento es un cliente en la fila. La cajera atiende de a uno y, mientras atiende a alguien, **nadie más avanza**. Si un cliente se pone a buscar monedas durante 5 minutos (un *slot* lento), toda la fila queda detenida, aunque los demás solo quieran pagar un chicle.

2.2 Una GUI congelada

¿Qué pasa si uno de esos métodos se demora? En el siguiente ejemplo hay un botón cuyo *slot* tarda 5 segundos (el `sleep` simula algo lento, como esperar un mensaje por la red o un cálculo pesado), y otro botón que solo cuenta clics.

1-event-loop/1_gui_congelada.py

```python
import sys
from time import sleep
from PyQt6.QtWidgets import QApplication, QWidget, QLabel, QPushButton


class MiVentana(QWidget):
    def __init__(self) -> None:
        super().__init__()
        self.clicks = 0
        self.init_gui()

    def init_gui(self) -> None:
        # Configuramos los widgets de la interfaz
        self.label = QLabel("Status: esperando", self)
        self.boton_lento = QPushButton("Tarea lenta (5 segundos)", self)
        self.boton_contador = QPushButton("Clicks: 0", self)

        self.label.setGeometry(10, 10, 280, 30)
        self.boton_lento.setGeometry(10, 50, 280, 30)
        self.boton_contador.setGeometry(10, 90, 280, 30)

        self.boton_lento.clicked.connect(self.tarea_lenta)
        self.boton_contador.clicked.connect(self.contar_click)
```

```python
        self.setWindowTitle("GUI congelada")
        self.setGeometry(50, 50, 300, 130)
        self.show()

    def tarea_lenta(self) -> None:
        # Se ejecuta en el thread principal, el mismo del event loop.
        # Mientras duerme, el event loop no procesa NINGÚN otro evento.
        self.label.setText("Status: trabajando...")  # ¿Alcanza a verse?
        sleep(5)
        self.label.setText("Status: tarea lenta terminada")

    def contar_click(self) -> None:
        self.clicks += 1
        self.boton_contador.setText(f"Clicks: {self.clicks}")

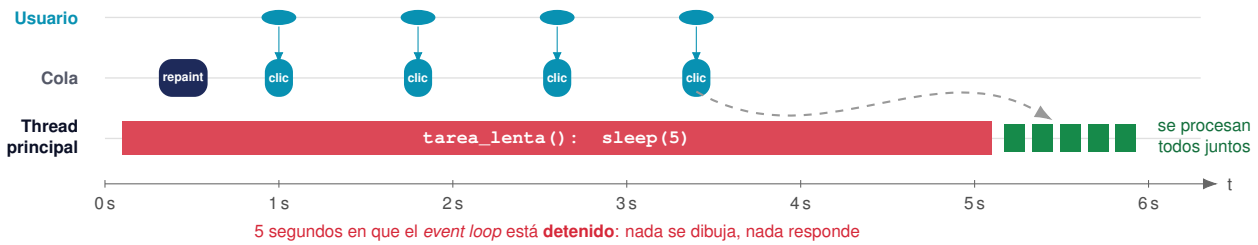

if __name__ == "__main__":
    def hook(type_, value, traceback):
        sys.__excepthook__(type_, value, traceback)

    sys.excepthook = hook
    app = QApplication([])
    ventana = MiVentana()
    sys.exit(app.exec())
```

Qué verás

Presiona "Tarea lenta" y luego varias veces el botón de clics. Vas a notar **tres cosas**:

1. **La ventana no responde** durante 5 segundos: no puedes moverla ni redimensionarla, y el sistema operativo puede incluso ofrecerte forzar su cierre.
2. **"Status: trabajando..." nunca aparece.** `setText` no dibuja en el acto: solo *avisa* que el *label* debe redibujarse, y ese aviso es **un evento más en la cola**. Se procesa cuando `tarea_lenta` termina... y para entonces el texto ya cambió a "terminada".
3. **Los clics no se pierden**: quedan en la cola y, apenas termina la tarea, se procesan todos juntos y el contador **salta de golpe**.



5 segundos en que el *event loop* está **detenido**: nada se dibuja, nada responde

Conclusión

Mientras un *slot* se está ejecutando, el *event loop* está detenido. Por eso todo método que corra en el *thread* principal debe **terminar rápido**.

2.3 ¿Y si usamos un *thread*?

La solución natural, después de la semana de *threading*, es mover el trabajo lento a otro *thread*. Pero aparece una pregunta nueva: si ese *thread* quiere mostrar algo, **¿puede modificar los *widgets* directamente?**

Hagamos el experimento. El siguiente programa crea **100 *threads***, cada uno con su propio `QLabel` de color al azar, y cada *thread* mueve **directamente** su *label* con `label.move()` cada 0,1 segundos. Para crear los *threads* usamos `QThread`, la clase de *thread* de PyQt; por ahora basta saber que se usa igual que `threading.Thread` (heredar, definir `run`, llamar a `start()`). La veremos a fondo en la Sección 4.

1-event-loop/2_sin_signal.py (fragmento: el thread)

```python
from random import randint
from time import sleep
from PyQt6.QtCore import QThread
from PyQt6.QtWidgets import QLabel


class Cuadrado(QThread):
    identificador = 0

    def __init__(self, label: QLabel, limite_x: int, limite_y: int):
        super().__init__()
        self.id = Cuadrado.identificador
        Cuadrado.identificador += 1
        self.label = label            # ¡el thread guarda el widget!
        self._posicion = [0, 0]
        self.posicion = [randint(0, limite_x), randint(0, limite_y)]

    @property
    def posicion(self) -> list:
        return self._posicion

    @posicion.setter
    def posicion(self, nueva_posicion: list) -> None:
        self._posicion = nueva_posicion
        nuevo_x, nuevo_y = self.posicion
        self.label.move(nuevo_x, nuevo_y)    # <- toca la interfaz DESDE el thread

    def run(self) -> None:
        while True:
            sleep(0.1)
            nuevo_x = self.posicion[0] + randint(-2, 2)
            nuevo_y = self.posicion[1] + randint(-2, 2)
            self.posicion = [nuevo_x, nuevo_y]
```

■ Qué verás (o no verás)

Lo más probable es que el programa **se caiga** con `Segmentation fault (core dumped)`. Otras veces no se moverá nada, o funcionará un rato y luego colapsará. **El resultado cambia en cada ejecución**: la marca típica de un problema de concurrencia.

2.3.1 ¿Por qué falla?

Los *widgets* de Qt **no son *thread-safe***: están diseñados para usarse **solo desde el *thread* principal**. Mientras un *thread* secundario cambia la posición de un *label*, el *event loop* puede estar **al mismo tiempo** dibujando ese mismo *label*, y el estado interno de Qt queda corrupto.

▲ Esto no se arregla con un `Lock`

No es la típica *race condition* entre dos *threads* nuestros. El otro participante es el *event loop*, que es **código interno de Qt (C++)** y no sabe nada de nuestros *locks*. No hay forma de "sincronizarse" con él: simplemente **no se tocan *widgets* desde otro *thread***.

2.4 La regla de oro

★ La regla de oro ★

Solo el *thread* principal toca la interfaz.

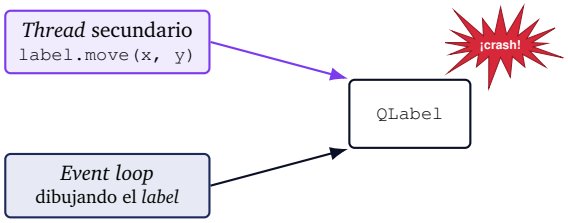
Los demás *threads* le **avisan** mediante **señales**.

¿Por qué las señales sí funcionan? Cuando un *thread* secundario emite una señal conectada a un método de un objeto que vive en el *thread* principal, PyQt **no ejecuta ese método en el *thread* que emitió**. En vez de eso, **pone un evento en la cola** del *thread* principal. Luego el *event loop* lo saca y ejecuta el método, igual que si fuera un clic.

Conexión encolada (*queued connection*)

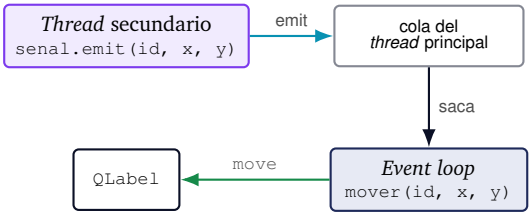
Cuando emisor y receptor están en *threads* distintos, `emit` **no llama** al *slot*: deja un **evento en la cola** del *thread* donde vive el receptor. El *slot* se ejecuta después, en **ese** *thread*. Así, el *thread* secundario nunca toca un *widget*: solo deja un "recado".

✗ Sin señal: el *thread* toca el *widget*



dos *threads* sobre el mismo *widget* a la vez
⇒ estado corrupto, `Segmentation fault`

✓ Con señal: conexión encolada



solo el *thread* principal modifica el *widget*; el secundario deja un "recado"

Este es el mismo experimento, pero ahora cada *thread* **emite una señal** con su nueva posición, y es la **ventana** quien mueve el *label*:

1-event-loop/3_con_signal.py

```python
class Cuadrado(QThread):
    identificador = 0

    def __init__(self, senal_mover: pyqtSignal, limite_x: int, limite_y: int) -> None:
        super().__init__()
        self.id = Cuadrado.identificador
        Cuadrado.identificador += 1
        self.senal_mover = senal_mover        # guardamos la SEÑAL, no el label
        self._posicion = [0, 0]
        self.posicion = (randint(0, limite_x), randint(0, limite_y))

    @property
    def posicion(self) -> list:
        return self._posicion

    @posicion.setter
    def posicion(self, nueva_posicion: list) -> None:
        self._posicion = nueva_posicion
        nuevo_x, nuevo_y = self.posicion
        self.senal_mover.emit(self.id, nuevo_x, nuevo_y)    # avisamos, no tocamos

    def run(self):
        while True:
            sleep(0.1)
            nuevo_x = self.posicion[0] + randint(-2, 2)
            nuevo_y = self.posicion[1] + randint(-2, 2)
            self.posicion = (nuevo_x, nuevo_y)


class MiVentana(QWidget):
    senal_mover = pyqtSignal(int, int, int)

    def __init__(self) -> None:
        super().__init__()
        self.setWindowTitle("Correcto uso de señales")
        self.setGeometry(200, 200, 500, 500)
        self.fondo = QLabel(self)
        self.fondo.setStyleSheet("background: orange")
        self.fondo.setGeometry(0, 0, 500, 500)

        self.cuadrados = []
        self.labels = {}
        for _ in range(100):
            self.crear_cuadrado()

        self.senal_mover.connect(self.mover)
        self.show()
```

```python

    def crear_cuadrado(self) -> None:
        label = QLabel(self)
        label.setGeometry(-50, -50, 50, 50)
        pixmap = QPixmap(50, 50)
        pixmap.fill(QColor(randint(20, 200), randint(20, 200), randint(20, 200)))
        label.setPixmap(pixmap)
        label.show()

        nuevo_cuadrado = Cuadrado(self.senal_mover, self.width(), self.height())
        self.labels[nuevo_cuadrado.id] = label
        self.cuadrados.append(nuevo_cuadrado)
        nuevo_cuadrado.start()

    def mover(self, id: int, x: int, y: int) -> None:
        # Se ejecuta en el thread PRINCIPAL: aquí sí se puede tocar la interfaz
        self.labels[id].move(x, y)
```

Qué verás

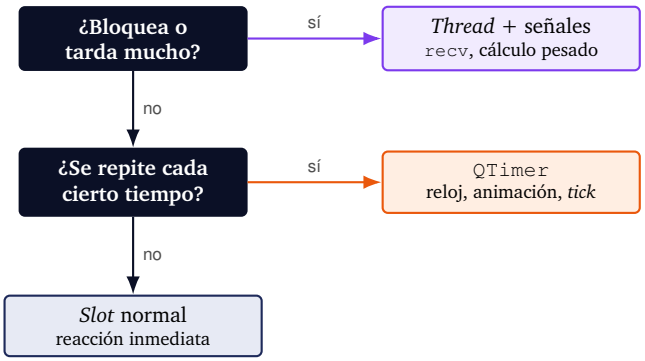
Los 100 cuadrados se mueven sin problemas. Los *threads* siguen haciendo su trabajo (calcular la nueva posición y esperar), pero **quien modifica la interfaz es siempre el *thread* principal**, en el método `mover`.

¿Qué cambió realmente? Solo dos cosas: el *thread* recibe una **señal** en vez del *label*, y la línea `self.label.move(...)` pasó a ser `self.senal_mover.emit(...)`. El diccionario `self.labels` (con `id` como llave) permite que un único *slot* sepa **qué** *label* mover.

2.5 ¿Necesito un *thread*?

Antes de crear un *thread*, pregúntate si de verdad lo necesitas. Esta es la pregunta que guía toda la semana:

¿Qué quiero hacer?	Ejemplo	Herramienta
Reaccionar a una acción del usuario, y la reacción es **rápida**	Cambiar un texto al hacer clic	Un *slot* normal. **Nada más.**
Algo que se **repite** cada cierto tiempo, y cada repetición es **rápida**	Mover personajes, un reloj, una animación, un temporizador	`QTimer` (Sección 3). Sin *threads* y **sin problemas de concurrencia**.
Algo que **bloquea** o que tarda mucho	Esperar mensajes con `recv`, un cálculo de varios segundos	Un *thread* (`QThread` o `threading.Thread`) que **avisa con señales** (Secciones 4 y 5).



Señal de alerta: el `sleep`.
Si escribes un `sleep` dentro de un *slot*, vas a congelar la interfaz. Y si lo usas para "esperar hasta el siguiente paso" (mover un personaje cada 50 ms), lo que quieres es un **`QTimer`**.

2.6 *Debuggeo*: el *hook* de excepciones

En PyQt6, si ocurre una excepción no capturada dentro de un *slot*, el programa puede **cerrarse completamente** (**`Aborted (core dumped)`**), aunque el error sea menor. Para que siga funcionando y solo imprima el error, instalamos nuestro propio *hook* en `sys.excepthook`. **Todos los *scripts* de la semana lo incluyen**:

(al final de cada script)

```python
if __name__ == "__main__":

    def hook(type_, value, traceback):
        # Imprimimos el error con el hook original de Python. Al instalar
        # nuestro propio sys.excepthook, PyQt no cierra el programa
        sys.__excepthook__(type_, value, traceback)

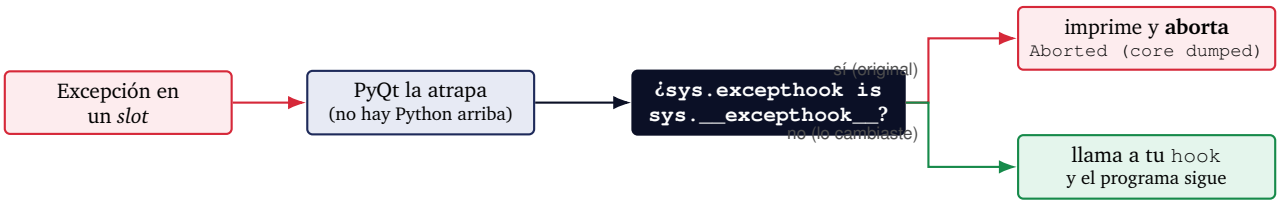
    sys.excepthook = hook
```

`sys.__excepthook__` guarda el *hook* **original** de Python (el que imprime el *traceback* completo). Nuestro `hook` simplemente lo llama.

2.6.1 ¿Por qué funciona, si `hook` hace lo mismo que el original?

La clave: **PyQt revisa *si cambiaste* `sys.excepthook`, no *qué hace* tu función.**

1. `sys.__excepthook__` es una **copia de respaldo** del *hook* original, y `sys.excepthook` es el que Python usa de verdad. En un Python sin modificar, al iniciar **son la misma función**: `sys.excepthook is sys.__excepthook__` es `True`.
2. Una excepción dentro de un *slot* **no tiene a dónde subir**: quien llamó al *slot* fue el *event loop*, que es código C++ de Qt, no Python. Entonces PyQt atrapa la excepción y decide qué hacer.
3. Si `sys.excepthook` **sigue siendo el original**, PyQt asume que nadie se hace cargo: imprime el *traceback* y **aborta** a propósito, para que el error no pase desapercibido.
4. Si `sys.excepthook` **es otra función**, PyQt asume que tú te haces cargo: la llama y **devuelve el control** al *event loop*. El programa sigue.



Instalación	Ante un error en un *slot*
`sys.excepthook = sys.__excepthook__`	Se cae con **`Aborted (core dumped)`** (es la misma función original).
`sys.excepthook = hook` (hook solo llama al original)	Se imprime el error y el programa **sigue funcionando**.

★ ¿Y por qué en mi Linux no se cae aunque no instale nada?

Algunas instalaciones de Python ya traen `sys.excepthook` reemplazado por otra función; PyQt ve que "no es el original" y sigue de largo. En Windows o macOS, en cambio, el mismo código **sí** se cae. Por eso instalamos el *hook* en todos los *scripts*: así el comportamiento es **el mismo en cualquier computador**. Ojo: el cierre no está dentro de `sys.__excepthook__` (que solo imprime); lo agrega PyQt cuando ve que el *hook* es el original.

▲ Que siga funcionando no significa que quedó bien

El *slot* que falló se interrumpió **a la mitad**, así que pudo dejar datos o la interfaz en un estado inconsistente. El *hook* sirve para **ver el error y seguir probando**, no para ignorarlo.

✓ Resumen express: *event loop* y regla de oro

- El *event loop* saca eventos de una cola y ejecuta sus *slots* **de a uno**, todo en el ***thread* principal**. Mientras un *slot* corre, **nada más avanza** (ni siquiera el redibujo).
- Un *slot* lento (`sleep`, `recv`, cálculo pesado) **congela** la GUI; los eventos no se pierden, se acumulan.

- Los *widgets* **no son *thread-safe***: tocarlos desde otro *thread* produce `Segmentation fault` o comportamientos aleatorios, y un `Lock` no lo arregla.
- **Regla de oro**: solo el *thread* principal toca la interfaz; los demás emiten señales (conexión encolada).
- Rápido → *slot*; periódico y rápido → `QTimer`; bloquea o tarda → *thread* + señales.
- Instala siempre `sys.excepthook = hook` para ver los errores sin que el programa se cierre.

3 `QTimer`: lo periódico, sin *threads*

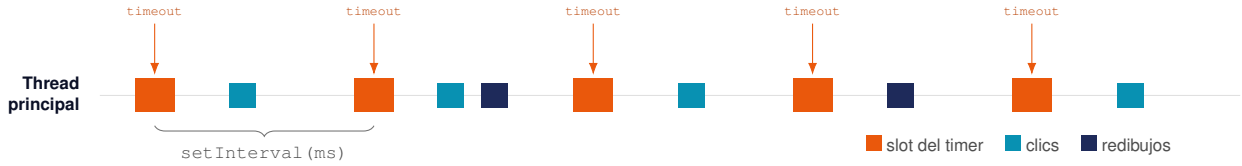
3.1 ¿Qué es un `QTimer`?

`QTimer` (módulo `PyQt6.QtCore`)

Un `QTimer` ejecuta un método **cada cierto intervalo de tiempo, dentro del *event loop***. Cuando se cumple el intervalo, emite su señal `timeout`, y el *event loop* ejecuta el método conectado **en el *thread* principal**, igual que un clic.

Se parece al `threading.Timer` de la semana de *threading*, pero con dos diferencias clave:

- **No crea ningún *thread*.** El método conectado a `timeout` corre en el *thread* principal, así que **sí puede modificar la interfaz** y **no hay problemas de concurrencia**.
- **Se repite**: por defecto vuelve a ejecutarse cada vez que se cumple el intervalo, hasta que lo detengamos.



los `timeout` se **intercalan** con los demás eventos en el mismo *thread*: cada repetición debe ser **rápida**

▲ Corre en el *thread* principal: debe ser rápido

Como el método conectado a `timeout` es un *slot* más, si demora **congela la interfaz**. Un `QTimer` es ideal para "mover 3 píxeles cada 16 ms", no para "descargar un archivo cada 1 segundo".

Método o señal	Descripción
`QTimer(parent)`	Crea el *timer*. Usar un *parent* (por ejemplo `self`) asegura que el objeto no sea eliminado mientras se usa.
`timeout`	**Señal** que se emite cada vez que se cumple el intervalo.
`setInterval(ms)`	Define el intervalo, en **milisegundos** (1000 ms = 1 s).
`start() / start(ms)`	Inicia (o reinicia) el *timer*. La segunda forma también define el intervalo.
`stop()`	Detiene el *timer*.
`isActive()`	`True` si el *timer* está corriendo.
`setSingleShot(True)`	Hace que el *timer* se ejecute **una sola vez**.
`remainingTime()`	Milisegundos que faltaban para el próximo `timeout`.
`QTimer.singleShot(ms, metodo)`	Método **estático**: ejecuta `metodo` una vez después de `ms`, sin crear un objeto.

★ La receta en 4 líneas

`timer = QTimer(self)` → `timer.timeout.connect(self.metodo)` → `timer.setInterval(ms)` → `timer.start()`. Fíjate: se conecta `self.metodo` **sin paréntesis**,

igual que con un botón.

3.2 Reloj digital

Un reloj que actualiza la hora cada 1 segundo:

2-qtimer/1_reloj_digital.py

```python
from PyQt6.QtWidgets import QApplication, QWidget, QLabel, QVBoxLayout
from PyQt6.QtCore import QTimer
from PyQt6.QtGui import QFont
import sys
import datetime


class RelojDigital(QWidget):
    def __init__(self) -> None:
        super().__init__()
        # Label encargado de mostrar la hora
        self.label_timer = QLabel()
        self.label_timer.setFont(QFont("Times", 50))

        layout = QVBoxLayout()
        layout.addWidget(self.label_timer)
        self.setLayout(layout)

        # QTimer encargado de actualizar el tiempo cada 1 segundo
        timer = QTimer(self)
        timer.timeout.connect(self.mostrar_hora)
        timer.setInterval(1000)
        timer.start()

        self.setWindowTitle("Reloj Digital con QTimer")
        self.setGeometry(100, 100, 250, 100)

        # Mostramos la hora por primera vez (si no, el label parte vacío 1 s)
        self.mostrar_hora()
        self.show()

    def mostrar_hora(self) -> None:
        hora_actual = datetime.datetime.now().time()
        self.label_timer.setText(hora_actual.strftime("%H:%M:%S"))
```

Qué verás



Una ventana con la hora actual que avanza cada segundo. **Detalle importante**: `timer` es una variable *local*, pero no desaparece al terminar `__init__` porque le dimos `self` como *parent*: la ventana lo mantiene vivo. Por eso se recomienda siempre `QTimer(self)`.

3.3 Múltiples `QTimer`

Podemos tener varios `QTimer` funcionando a la vez, cada uno con su intervalo. Aquí, cada `MiTimer` es un `QObject` con su propio `QTimer` que avisa a la ventana con una señal. Al presionar el botón se crean cinco, con intervalos de 0,1 a 0,5 segundos; cada uno cuenta hasta 9 y se detiene. (En la Sección 4 haremos **el mismo ejemplo con *threads***: compáralos.)

2-qtimer/2_multiples_qtimers.py

```python
import sys
from PyQt6.QtCore import pyqtSignal, QObject, QTimer
from PyQt6.QtWidgets import QApplication, QWidget, QLabel, QPushButton


class MiTimer(QObject):
    senal_actualizar = pyqtSignal(int, str)

    def __init__(self, indice: int, tiempo: float) -> None:
```

```python
        super().__init__()
        self.indice = indice
        self.tiempo = tiempo
        self.indice_actual = 0
        self.timer = QTimer(self)
        self.timer.setInterval(int(tiempo * 1000))    # segundos -> milisegundos
        self.timer.timeout.connect(self.enviar_dato)

    def enviar_dato(self) -> None:
        if self.indice_actual <= 9:
            self.senal_actualizar.emit(self.indice, str(self.indice_actual))
            self.indice_actual += 1
        else:
            self.senal_actualizar.emit(self.indice, "Status: Qtimer terminado")
            self.timer.stop()

    def comenzar(self) -> None:
        self.timer.start()

    def sigue_andando(self) -> bool:
        return self.timer.isActive()

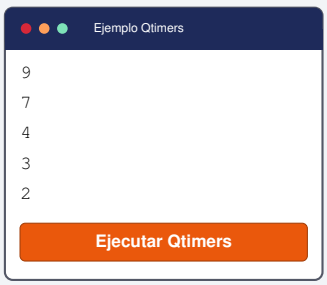

class MiVentana(QWidget):
    def __init__(self) -> None:
        super().__init__()
        self.timers = []
        self.init_gui()

    def init_gui(self) -> None:
        # Un label por cada QTimer
        self.labels = {i: QLabel("Status: esperando Qtimer", self) for i in range(1, 6)}
        self.boton = QPushButton("Ejecutar Qtimers", self)
        self.boton.clicked.connect(self.ejecutar_timers)
        for i in range(1, 6):
            self.labels[i].setGeometry(10, (i - 1) * 30, 230, 30)
        self.boton.setGeometry(10, 150, 230, 30)
        self.setWindowTitle("Ejemplo Qtimers")
        self.setGeometry(50, 50, 250, 200)
        self.show()

    def ejecutar_timers(self) -> None:
        # Si alguno sigue andando, no creamos otros
        for timer in self.timers:
            if timer.sigue_andando():
                return
        self.timers = []
        for i in range(1, 6):
            timer = MiTimer(i, i / 10)
            timer.senal_actualizar.connect(self.actualizar_labels)
            self.timers.append(timer)
            timer.comenzar()

    def actualizar_labels(self, indice: int, texto: str) -> None:
        self.labels[indice].setText(texto)
```

Qué verás



Los cinco contadores avanzan a distinta velocidad: el primero (0,1 s) llega a 9 rápido y muestra "Status: Qtimer terminado", mientras el quinto (0,5 s) va lento. Si presionas el botón mientras alguno corre, no pasa nada. **Todo ocurre en el *thread* principal**: los cinco `timeout` se van intercalando en la cola.

¿Por qué `MiTimer` hereda de `QObject`? Porque define una `pyqtSignal`, y las señales **solo pueden definirse en clases que heredan de `QObject`** (como vimos la semana pasada). Además, `self.timer = QTimer(self)` usa al propio `MiTimer` como *parent*.

3.4 Una sola vez: `singleShot`

A veces queremos ejecutar algo **una sola vez** después de cierto tiempo: mostrar un mensaje por 3 segundos, hacer aparecer un enemigo más tarde, etc. Hay dos formas:

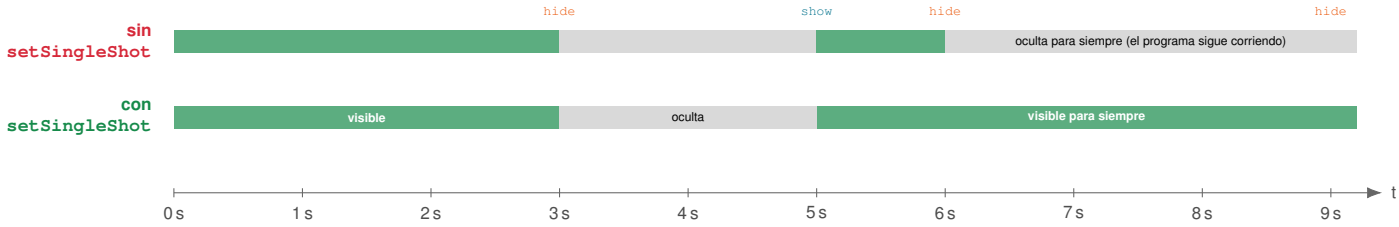
Forma	Código	Cuándo
1. Objeto con `setSingleShot(True)`	`t = QTimer(self)` `t.setSingleShot(True)`	Si después lo quieres **detener o reiniciar**.
2. Método estático	`QTimer.singleShot(5000, self.show)`	Más corto, pero **no se puede cancelar**.

En este ejemplo (que extiende el reloj), la ventana se esconde a los 3 segundos (forma 1) y vuelve a aparecer a los 5 (forma 2):

2-qtimer/3_singleshot.py (fragmento del __init__)

```python
        # Forma 1: un QTimer configurado con setSingleShot(True).
        # Después de 3 segundos esconde la ventana, y no se vuelve a ejecutar
        self.timer_esconder = QTimer(self)
        self.timer_esconder.setSingleShot(True)
        self.timer_esconder.timeout.connect(self.hide)
        self.timer_esconder.setInterval(3000)
        self.timer_esconder.start()

        # Forma 2: el método estático QTimer.singleShot, sin crear un objeto.
        # Después de 5 segundos vuelve a mostrar la ventana
        QTimer.singleShot(5000, self.show)
```

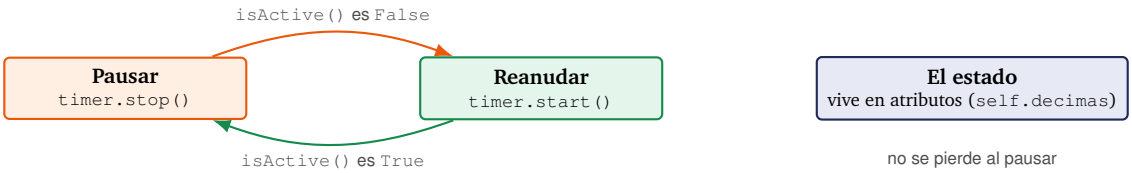


▲ Experimento: comenta `setSingleShot(True)`

El *timer* volverá a esconder la ventana **cada 3 segundos**. Como `QTimer.singleShot` la muestra una sola vez (a los 5 s), desde los 6 s la ventana queda escondida para siempre, pero **el programa sigue corriendo**. Para terminarlo, cierra la terminal o presiona `Ctrl+\` (Linux y macOS). `Ctrl+C` **no sirve**: la interrupción llega como una excepción dentro de un *slot*, y nuestro *hook* la imprime y deja el programa funcionando.

3.5 Pausar y reanudar

Un `QTimer` es el "motor" que hace avanzar algo en el tiempo. Entonces:



2-qtimer/4_pausa.py

```python
class Cronometro(QWidget):
    def __init__(self) -> None:
        super().__init__()
        self.decimas = 0                        # décimas de segundo transcurridas
        # Este QTimer avanza el cronómetro cada 100 ms. No lo iniciamos aún
        self.timer = QTimer(self)
        self.timer.setInterval(100)
```

```python
        self.timer.timeout.connect(self.avanzar)
        self.init_gui()

    def init_gui(self) -> None:
        self.label = QLabel(self)
        self.label.setFont(QFont("Monospace", 30))
        self.label.setGeometry(10, 10, 230, 50)

        self.boton_pausa = QPushButton("Iniciar", self)
        self.boton_pausa.setGeometry(10, 70, 110, 30)
        self.boton_pausa.clicked.connect(self.pausar_o_reanudar)

        self.boton_reiniciar = QPushButton("Reiniciar", self)
        self.boton_reiniciar.setGeometry(130, 70, 110, 30)
        self.boton_reiniciar.clicked.connect(self.reiniciar)

        self.setWindowTitle("Cronómetro con QTimer")
        self.setGeometry(50, 50, 250, 110)
        self.actualizar_label()
        self.show()

    def avanzar(self) -> None:
        self.decimas += 1
        self.actualizar_label()

    def pausar_o_reanudar(self) -> None:
        # Pausar = detener el QTimer; reanudar = volver a iniciarlo.
        # El estado (self.decimas) no se pierde
        if self.timer.isActive():
            self.timer.stop()
            self.boton_pausa.setText("Reanudar")
        else:
            self.timer.start()
            self.boton_pausa.setText("Pausar")

    def reiniciar(self) -> None:
        self.timer.stop()
        self.decimas = 0
        self.boton_pausa.setText("Iniciar")
        self.actualizar_label()

    def actualizar_label(self) -> None:
        minutos, decimas = divmod(self.decimas, 600)
        self.label.setText(f"{minutos:02d}:{decimas / 10:04.1f}")
```

■ Qué verás



"Iniciar" arranca el conteo; el botón cambia a "Pausar" y luego a "Reanudar". "Reiniciar" detiene el *timer* y vuelve a `00:00.0`. `divmod(self.decimas, 600)` separa minutos (600 décimas) del resto.

★ Dos detalles para tus tareas

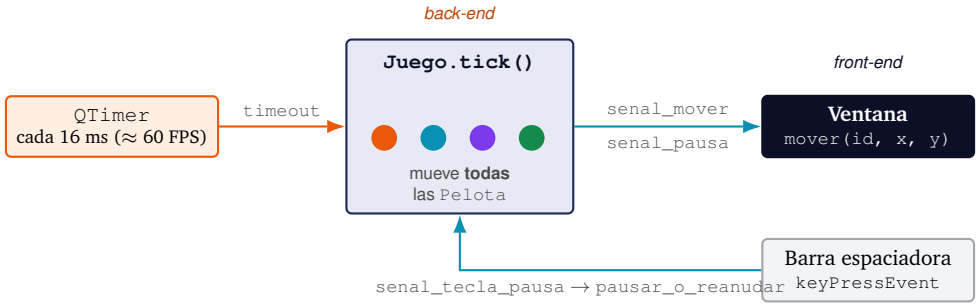
- **En un juego**, pausar significa detener **todos** los *timers* que lo hacen avanzar. Tenlos accesibles, por ejemplo en una **lista**.
- Al reanudar con `start()`, **el intervalo parte de cero**. Con 100 ms no se nota, pero si un *timer* de 10 s se pausa cuando le faltaba 1 s, al reanudar esperará 10 s completos. Si importa, `remainingTime()` te dice cuántos ms faltaban **antes** de detenerlo.

3.6 *Loop* de juego con un solo `QTimer`

En un juego o simulación con muchos elementos en movimiento, una opción muy cómoda es tener **un único** `QTimer` (el ***tick*** del juego) que en cada ejecución actualiza a **todos** los elementos. Comparado con un *timer* o un *thread* por elemento:

- Todos los elementos avanzan **en el mismo instante**: es fácil detectar colisiones o comparar posiciones.

- **Pausar** es detener **un solo** *timer*.
- Todo ocurre en el *thread* principal: **cero** problemas de concurrencia.



En el ejemplo, 30 pelotas rebotan dentro de la ventana. `Pelota` es una **clase común** (ni *thread* ni `QObject`): solo guarda posición y velocidad. El *back-end* (`Juego`) las mueve todas en cada *tick* y avisa al *front-end* (`Ventana`) con señales. La barra espaciadora pausa y reanuda.

2-qtimer/5_loop_de_juego.py

```python
import sys
from random import randint
from PyQt6.QtCore import QObject, QTimer, Qt, pyqtSignal
from PyQt6.QtGui import QColor, QKeyEvent, QPixmap
from PyQt6.QtWidgets import QApplication, QWidget, QLabel

ANCHO, ALTO = 500, 400
TAMAÑO = 20
CANTIDAD_PELOTAS = 30
TICK_MS = 16  # ~60 actualizaciones por segundo


class Pelota:
    """Solo datos: posición y velocidad. Quien la mueve es el Juego."""

    def __init__(self, id_pelota: int) -> None:
        self.id = id_pelota
        self.x = randint(0, ANCHO - TAMAÑO)
        self.y = randint(0, ALTO - TAMAÑO)
        self.vx = randint(-4, 4) or 1      # "or 1": evita velocidad 0
        self.vy = randint(-4, 4) or 1

    def mover(self) -> None:
        self.x += self.vx
        self.y += self.vy
        # Rebotamos en los bordes de la ventana
        if not 0 <= self.x <= ANCHO - TAMAÑO:
            self.vx *= -1
        if not 0 <= self.y <= ALTO - TAMAÑO:
            self.vy *= -1


class Juego(QObject):
    """Back-end. Un único QTimer hace avanzar a TODAS las pelotas."""
    senal_mover = pyqtSignal(int, int, int)  # id, x, y
    senal_pausa = pyqtSignal(bool)           # True si quedó en pausa

    def __init__(self) -> None:
        super().__init__()
        self.pelotas = [Pelota(i) for i in range(CANTIDAD_PELOTAS)]
        self.timer = QTimer(self)
        self.timer.setInterval(TICK_MS)
        self.timer.timeout.connect(self.tick)

    def iniciar(self) -> None:
        self.timer.start()

    def tick(self) -> None:
        # Actualizamos TODO el juego y avisamos al front-end.
        # Todo ocurre en el thread principal: no hay concurrencia que cuidar
        for pelota in self.pelotas:
```

```python
            pelota.mover()
            self.senal_mover.emit(pelota.id, pelota.x, pelota.y)

    def pausar_o_reanudar(self) -> None:
        if self.timer.isActive():
            self.timer.stop()
        else:
            self.timer.start()
        self.senal_pausa.emit(not self.timer.isActive())


class Ventana(QWidget):
    """Front-end. Solo dibuja y avisa cuando se presiona la barra espaciadora."""
    senal_tecla_pausa = pyqtSignal()

    def __init__(self) -> None:
        super().__init__()
        self.setWindowTitle("Loop de juego con un QTimer (Espacio: pausa)")
        self.setFixedSize(ANCHO, ALTO)
        self.setStyleSheet("background: #1e1e2e;")
        self.labels = {}
        for i in range(CANTIDAD_PELOTAS):
            pixmap = QPixmap(TAMAÑO, TAMAÑO)
            pixmap.fill(QColor(randint(80, 255), randint(80, 255), randint(80, 255)))
            label = QLabel(self)
            label.setPixmap(pixmap)
            label.setGeometry(-TAMAÑO, -TAMAÑO, TAMAÑO, TAMAÑO)
            self.labels[i] = label
        self.label_pausa = QLabel("PAUSA", self)
        self.label_pausa.setStyleSheet("color: white; font-size: 40px;")
        self.label_pausa.setGeometry(ANCHO // 2 - 70, ALTO // 2 - 30, 140, 60)
        self.label_pausa.hide()

    def mover(self, id_pelota: int, x: int, y: int) -> None:
        self.labels[id_pelota].move(x, y)

    def mostrar_pausa(self, en_pausa: bool) -> None:
        self.label_pausa.setVisible(en_pausa)

    def keyPressEvent(self, evento: QKeyEvent) -> None:
        # isAutoRepeat(): solo reaccionamos cuando la tecla SE ACABA de presionar
        if evento.key() == Qt.Key.Key_Space and not evento.isAutoRepeat():
            self.senal_tecla_pausa.emit()


if __name__ == "__main__":
    def hook(type_, value, traceback):
        sys.__excepthook__(type_, value, traceback)

    sys.excepthook = hook
    app = QApplication([])
    juego = Juego()
    ventana = Ventana()
    # Conectamos back-end y front-end
    juego.senal_mover.connect(ventana.mover)
    juego.senal_pausa.connect(ventana.mostrar_pausa)
    ventana.senal_tecla_pausa.connect(juego.pausar_o_reanudar)
    ventana.show()
    juego.iniciar()
    sys.exit(app.exec())
```

Qué verás



30 cuadrados de colores rebotan por la ventana. Con la barra espaciadora todos se congelan a la vez y aparece "PAUSA"; otra vez y siguen exactamente donde estaban. Si mantienes la barra presionada, **no** parpadea entre pausa y juego gracias a `isAutoRepeat()` (Sección 7).

Fíjate en que `Ventana` **no sabe nada del `QTimer`** ni de cómo se mueven las pelotas: solo dibuja lo que le dice el *back-end* y le avisa cuando se presiona la barra. Es la misma separación *front-end/back-end* de la semana pasada.

✓ Resumen express: `QTimer`

- `QTimer` ejecuta un método cada `setInterval(ms)` **en el *thread* principal**: no crea *threads*, puede tocar la interfaz y no hay concurrencia.
- Receta: `QTimer(self)` → `timeout.connect(metodo)` → `setInterval(ms)` → `start()`. Detener con `stop()`, consultar con `isActive()`.
- Una sola vez: `setSingleShot(True)` (cancelable) o `QTimer.singleShot(ms, metodo)` (no cancelable).
- Pausar = `stop()`; reanudar = `start()` (el intervalo parte de cero; ver `remainingTime()`).
- Para juegos: **un solo `QTimer`** que actualiza todo en cada *tick*. Cada *tick* debe ser **rápido**.

4 `QThread`: lo que bloquea, con señales

4.1 *Threads* y PyQt

Cuando una tarea **bloquea** o **tarda mucho**, la ejecutamos en otro *thread*. Todo lo de la semana de *threading* sigue siendo válido: podemos usar `threading.Thread` junto con PyQt, **siempre que respetemos la regla de oro**: el *thread* no toca la interfaz, sino que **emite señales**.

En el siguiente ejemplo la ventana define una señal y **se la entrega** al *thread*. El *thread* la emite 10 veces, y la ventana actualiza su *label* en el método conectado (que corre en el *thread* principal).

3-qthread/1_thread.py

```python
import sys
from threading import Thread
from time import sleep
from PyQt6.QtCore import pyqtSignal
from PyQt6.QtWidgets import QApplication, QWidget, QLabel, QPushButton


class MiThread(Thread):
    def __init__(self, senal_actualizar: pyqtSignal) -> None:
        super().__init__()
        self.senal_actualizar = senal_actualizar   # recibe la señal de la ventana

    def run(self) -> None:
        for i in range(10):
            sleep(0.5)
            self.senal_actualizar.emit(str(i))
        self.senal_actualizar.emit("Status: thread terminado")


class MiVentana(QWidget):
    # La señal se define en la VENTANA (un QObject): Thread no puede definirla
    senal_thread = pyqtSignal(str)

    def __init__(self) -> None:
```

```
25          super().__init__()
26          self.mi_thread = None
27          self.senal_thread.connect(self.actualizar_label)
28          self.init_gui()
29
30      def init_gui(self) -> None:
31          self.label = QLabel("Status: esperando thread", self)
32          self.boton = QPushButton("Ejecutar Thread", self)
33          self.boton.clicked.connect(self.ejecutar_thread)
34          self.label.setGeometry(10, 10, 230, 30)
35          self.boton.setGeometry(10, 50, 230, 30)
36          self.setWindowTitle("Ejemplo thread")
37          self.setGeometry(50, 50, 250, 200)
38          self.show()
39
40      def ejecutar_thread(self) -> None:
41          # Creamos un thread solo si no hay otro corriendo
42          if self.mi_thread is None or not self.mi_thread.is_alive():
43              self.mi_thread = MiThread(self.senal_thread)
44              self.mi_thread.start()
45
46      def actualizar_label(self, texto: str) -> None:
47          self.label.setText(texto)
```

■ Qué verás

Al presionar "Ejecutar Thread", el *label* muestra `0, 1, ..., 9` cada medio segundo y termina con "Status: thread terminado". **La ventana nunca se congela**: el `sleep` ocurre en el *thread* secundario, y la ventana sigue respondiendo.

4.2 `QThread`

PyQt ofrece su propia clase de *thread*: `QThread` (módulo `PyQt6.QtCore`). Se usa **igual** que `Thread`: heredar, definir `run` e iniciar con `start()`. La ventaja es que, al ser parte de PyQt, se integra mejor: puede **definir sus propias señales** y trae señales propias como `finished` (se emite cuando `run` termina).

threading.Thread	QThread	Nota
`start()`	**`start()`**	Inicia el *thread* (llama a `run` en el *thread* nuevo).
`run()`	**`run()`**	Lo que ejecuta el *thread*.
`join()`	**`wait()`**	Espera a que el *thread* termine.
`is_alive()`	**`isRunning()`**	¿Sigue corriendo?
`daemon = True`	No existe	Hay que **detenerlos nosotros** (Sección 4.5).

El ejemplo anterior con `QThread` solo cambia la herencia y `is_alive()` por `isRunning()`:

3-qthread/2_qthread.py (fragmento)

```
1  class MiThread(QThread):
2      def __init__(self, senal_actualizar: pyqtSignal) -> None:
3          super().__init__()
4          self.senal_actualizar = senal_actualizar
5
6      def run(self) -> None:
7          for i in range(10):
8              sleep(0.5)
9              self.senal_actualizar.emit(str(i))
10         sleep(0.5)
11         self.senal_actualizar.emit("Status: QThread terminado")
```

▲ ¡No llames `self.thread` a tu atributo!

Guardamos el *thread* en `self.mi_thread`, y **no** en `self.thread`. Todo objeto de PyQt (incluida la ventana) ya tiene un **método** `thread()` que retorna el *thread* donde vive el objeto. Si lo sobrescribes con un atributo, ese método deja de funcionar y PyQt se confunde.

4.2.1 `QThread` con su propia señal

Como `QThread` hereda de `QObject`, puede definir sus señales **como atributos de clase**. Esto **no es posible con `Thread`**. Así la ventana ya no necesita entregarle su señal: basta **conectar la señal del *thread*** después de crearlo.

3-qthread/3_qthread_con_signal.py

```python
import sys
from time import sleep
from PyQt6.QtCore import pyqtSignal, QThread
from PyQt6.QtWidgets import QApplication, QWidget, QLabel, QPushButton


class MiThread(QThread):
    senal_thread = pyqtSignal(str)          # ¡la señal es del propio thread!

    def __init__(self) -> None:
        super().__init__()

    def run(self) -> None:
        for i in range(10):
            sleep(0.5)
            self.senal_thread.emit(str(i))
        sleep(0.5)
        self.senal_thread.emit("Status: QThread terminado")


class MiVentana(QWidget):
    def __init__(self) -> None:
        super().__init__()
        self.mi_thread = None
        self.init_gui()

    def init_gui(self) -> None:
        self.label = QLabel("Status: esperando QThread", self)
        self.boton = QPushButton("Ejecutar QThread", self)
        self.boton.clicked.connect(self.ejecutar_thread)
        self.label.setGeometry(10, 10, 230, 30)
        self.boton.setGeometry(10, 50, 230, 30)
        self.setWindowTitle("Ejemplo QThread")
        self.setGeometry(50, 50, 250, 200)
        self.show()

    def ejecutar_thread(self) -> None:
        # Aquí debemos ocupar isRunning en lugar de is_alive
        if self.mi_thread is None or not self.mi_thread.isRunning():
            self.mi_thread = MiThread()
            self.mi_thread.senal_thread.connect(self.actualizar_label)
            self.mi_thread.start()

    def actualizar_label(self, texto: str) -> None:
        self.label.setText(texto)
```



4.3 Afinidad: ¿en qué *thread* se ejecuta cada método?

Este es el detalle de `QThread` que causa más errores:

Afinidad

Un objeto `QThread` no es el *thread*: es un objeto que *administra* un *thread*. Y como todo objeto de PyQt, "vive" en el *thread* **donde fue creado**. Eso se llama **afinidad**.

Dos consecuencias:

1. De un `QThread`, **solo el método `run` se ejecuta en el *thread* nuevo**. Cualquier otro método se ejecuta en otro lado: si lo llamas directo, en el *thread* que lo llamó (como cualquier función de Python); si está conectado a una señal, en el *thread* donde **vive el objeto** (normalmente, el principal).
2. Al emitirse una señal, el método conectado se ejecuta **en el *thread* donde vive el objeto receptor**, no en el que emitió. Si son distintos, la llamada se encola y la ejecuta el *event loop* del *thread* receptor.

Para verlo, el ejemplo imprime en qué *thread* corre cada método con dos funciones de `threading`: `current_thread()` (el *thread* que ejecuta el código) y `main_thread()` (el *thread* principal, el del *event loop*). Si son el mismo objeto, estamos en el principal. El botón "Pedirle una tarea pesada" emite una señal conectada al método `tarea_pesada` **del `QThread`**.

3-qthread/4_afinidad.py

```python
import sys
import threading
from time import sleep
from PyQt6.QtCore import pyqtSignal, QThread
from PyQt6.QtWidgets import QApplication, QWidget, QLabel, QPushButton


def donde_estoy() -> str:
    if threading.current_thread() is threading.main_thread():
        return "thread PRINCIPAL"
    return "thread SECUNDARIO"


class MiThread(QThread):
    senal_progreso = pyqtSignal(str)

    def run(self) -> None:
        # run es lo único que se ejecuta en el thread nuevo
        print(f"[run] me ejecuto en el {donde_estoy()}")
        for i in range(1, 6):
            sleep(1)
            self.senal_progreso.emit(f"Paso {i} de 5")
        self.senal_progreso.emit("Status: QThread terminado")

    def tarea_pesada(self) -> None:
        # Este método es un slot del QThread. ¿Dónde se ejecuta?
        print(f"[tarea_pesada] me ejecuto en el {donde_estoy()}")
        sleep(3)
        print("[tarea_pesada] terminé")


class MiVentana(QWidget):
    senal_pedir_tarea = pyqtSignal()

    def __init__(self) -> None:
        super().__init__()
        print(f"[__init__ de MiVentana] me ejecuto en el {donde_estoy()}")
        self.mi_thread = MiThread()        # se CREA en el thread principal
        self.mi_thread.senal_progreso.connect(self.actualizar_label)
        self.senal_pedir_tarea.connect(self.mi_thread.tarea_pesada)
        self.init_gui()

    def init_gui(self) -> None:
        self.label = QLabel("Status: esperando QThread", self)
        self.boton_iniciar = QPushButton("Iniciar QThread", self)
        self.boton_tarea = QPushButton("Pedirle una tarea pesada al QThread", self)
        self.label.setGeometry(10, 10, 330, 30)
        self.boton_iniciar.setGeometry(10, 50, 330, 30)
        self.boton_tarea.setGeometry(10, 90, 330, 30)
        self.boton_iniciar.clicked.connect(self.iniciar_thread)
        self.boton_tarea.clicked.connect(self.senal_pedir_tarea.emit)
        self.setWindowTitle("Afinidad de threads")
        self.setGeometry(50, 50, 350, 130)
        self.show()

    def iniciar_thread(self) -> None:
        if not self.mi_thread.isRunning():
            self.mi_thread.start()

```

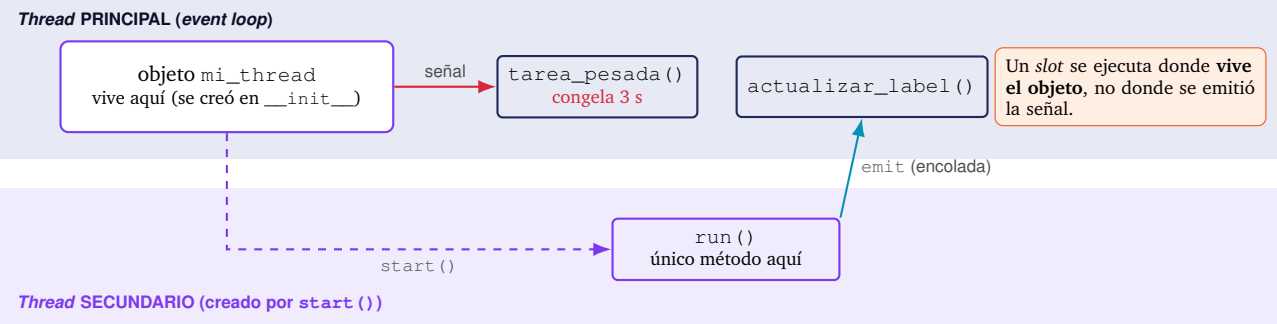
```python
    def actualizar_label(self, texto: str) -> None:
        print(f"[actualizar_label] me ejecuto en el {donde_estoy()}")
        self.label.setText(texto)
```

Al presionar “Iniciar QThread” y luego “Pedirle una tarea pesada”, la consola muestra:

```
>_ __init__ de MiVentana

 me ejecuto en el thread PRINCIPAL
[run] me ejecuto en el thread SECUNDARIO
[actualizar_label] me ejecuto en el thread PRINCIPAL
[tarea_pesada] me ejecuto en el thread PRINCIPAL     <-- ¡sorpresa!
[tarea_pesada] terminé
[actualizar_label] me ejecuto en el thread PRINCIPAL
...
```

`tarea_pesada` es un método del `QThread`, pero corrió en el *thread* **principal**, y la ventana **se congeló 3 segundos**. El objeto `MiThread` se creó en el `__init__` de la ventana, o sea en el *thread* principal, así que **vive ahí**.



★ Conclusión práctica

El trabajo pesado va dentro de `run`. Si necesitas pedirle cosas al *thread* mientras corre, usa **atributos que `run` revise** en su ciclo (una bandera, o una `Queue` como en la semana de *threading*), **no** métodos conectados a señales.

4.3.1 Precaución: ¿dónde se crea un objeto?

La afinidad explica un error más sutil. El código principal crea y ejecuta un `QThread` (T1), y **dentro del `run` de T1** se crea y ejecuta otro `QThread` (T2). Conectamos `signal_t1` desde el *thread* principal y `signal_t2` desde el *thread* de T1, y T1 y T2 emiten **ambas** señales. ¿Se ejecutan las 4?

3-qthread/5a_signals_no_funcionan.py

```python
import sys
from PyQt6.QtCore import QThread, QTimer, pyqtSignal
from PyQt6.QtWidgets import QApplication


class Thread1(QThread):
    signal_t1 = pyqtSignal(str)

    def __init__(self) -> None:
        super().__init__()
        self.t2 = None

    def run(self) -> None:
        print("[Thread 1] Crear Thread 2")
        # OJO: este objeto se crea DENTRO de run, es decir, en el thread de T1.
        # Por lo tanto, t2 "vive" (tiene afinidad) en el thread de T1
        self.t2 = Thread2(self.signal_t1)

        print("[Thread 1] Conectar signal_t2 con imprimir de T2")
        self.t2.signal_t2.connect(self.t2.imprimir)

        print("[Thread 1] Ejecutar Thread 2 y esperar a que termine")
```

```python
        self.t2.start()
        self.t2.wait()

        print("[Thread 1] Emitir signal_t1 y signal_t2")
        self.signal_t1.emit("signal_t1 emitida desde Thread 1")
        self.t2.signal_t2.emit("signal_t2 emitida desde Thread 1")

    def imprimir(self, texto: str) -> None:
        print("\t[imprimir de T1] ->", texto)


class Thread2(QThread):
    signal_t2 = pyqtSignal(str)

    def __init__(self, signal_t1: pyqtSignal) -> None:
        super().__init__()
        self.signal_t1 = signal_t1

    def run(self) -> None:
        print("[Thread 2] Emitir signal_t1 y signal_t2")
        self.signal_t1.emit("signal_t1 emitida desde Thread 2")
        self.signal_t2.emit("signal_t2 emitida desde Thread 2")

    def imprimir(self, texto: str) -> None:
        print("\t[imprimir de T2] ->", texto)


if __name__ == "__main__":
    def hook(type_, value, traceback):
        sys.__excepthook__(type_, value, traceback)

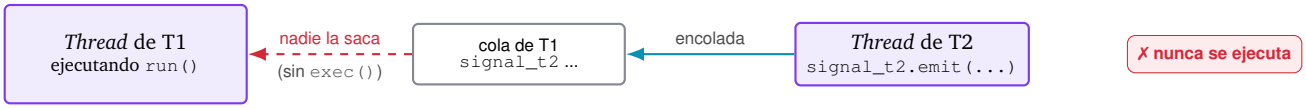
    sys.excepthook = hook
    app = QApplication([])
    print("[Main] Crear Thread 1")
    hilo = Thread1()
    print("[Main] Conectar signal_t1 con imprimir de T1")
    hilo.signal_t1.connect(hilo.imprimir)
    print("[Main] Ejecutar Thread 1")
    hilo.start()
    QTimer.singleShot(1000, app.quit)    # cerramos después de 1 segundo
    app.exec()
    hilo.wait()
```

>_ Terminal (el orden de las 2 últimas líneas puede variar)

```
[Main] Crear Thread 1
[Main] Conectar signal_t1 con imprimir de T1
[Main] Ejecutar Thread 1
[Thread 1] Crear Thread 2
[Thread 1] Conectar signal_t2 con imprimir de T2
[Thread 1] Ejecutar Thread 2 y esperar a que termine
[Thread 2] Emitir signal_t1 y signal_t2
    [imprimir de T1] -> signal_t1 emitida desde Thread 2
[Thread 1] Emitir signal_t1 y signal_t2
    [imprimir de T1] -> signal_t1 emitida desde Thread 1
    [imprimir de T2] -> signal_t2 emitida desde Thread 1
```

Se emitieron 4 señales, pero **solo se ejecutaron 3**: falta `signal_t2` emitida desde Thread 2. ¿Por qué?

Señal	Emite	Receptor vive en	¿Qué pasa?
`signal_t1`	T1 o T2	*thread* principal (`hilo` se creó ahí)	Se encola; el principal tiene *event loop* (`app.exec()`) y la ejecuta. ✓
`signal_t2`	T1	*thread* de T1	Mismo *thread*: `t2.imprimir` se ejecuta directo. ✓
`signal_t2`	T2	*thread* de T1	Se encola en T1... pero T1 **no tiene *event loop*** (ejecuta nuestro `run`, nunca llama a `exec()`). Nadie la saca de la cola. ✗



La solución es crear T2 **en el *thread* principal**, por ejemplo en el `__init__` de T1 (que se ejecuta en el principal). Con ese único cambio, se ejecutan las 4 señales:

3-qthread/5b_signals_corregido.py (fragmento)

```python
    def __init__(self) -> None:
        super().__init__()
        # CORRECCIÓN: creamos T2 en __init__, que se ejecuta en el thread
        # principal. Así t2 vive en el thread principal, que sí tiene un
        # event loop (app.exec()) para procesar sus señales
        self.t2 = Thread2(self.signal_t1)

    def run(self) -> None:
        print("[Thread 1] Conectar signal_t2 con imprimir de T2")
        self.t2.signal_t2.connect(self.t2.imprimir)
        print("[Thread 1] Ejecutar Thread 2 y esperar a que termine")
        self.t2.start()
        self.t2.wait()
        print("[Thread 1] Emitir signal_t1 y signal_t2")
        self.signal_t1.emit("signal_t1 emitida desde Thread 1")
        self.t2.signal_t2.emit("signal_t2 emitida desde Thread 1")
```

Regla práctica

Crea los objetos de PyQt (`QThread`, `QTimer`, `QObject` con señales, *widgets*) **en el *thread* principal**, por ejemplo en un `__init__` o en un *slot*, y **nunca dentro de `run`**. Por la misma razón, un `QTimer` creado dentro de `run` **no funciona**: su `timeout` necesita un *event loop* en ese *thread*.

4.4 Múltiples `QThread` y la referencia

Ahora queremos una cantidad variable de `QThread` corriendo a la vez. Lo más importante: **siempre guardar una referencia** a cada `QThread` mientras corre, por ejemplo en una lista. Es el mismo ejemplo de los cinco `QTimer`, ahora con *threads*:

3-qthread/6_multiples_qthreads.py (lo nuevo)

```python
class MiThread(QThread):
    # Se define en la clase; cada instancia tiene su propia señal
    senal_actualizar = pyqtSignal(int, str)

    def __init__(self, indice: int, tiempo: float) -> None:
        super().__init__()
        self.indice = indice
        self.tiempo = tiempo

    def run(self) -> None:
        for i in range(10):
            sleep(self.tiempo)
            self.senal_actualizar.emit(self.indice, str(i))
        sleep(self.tiempo)
        self.senal_actualizar.emit(self.indice, "Status: QThread terminado")


class MiVentana(QWidget):
    def __init__(self) -> None:
        super().__init__()
        self.threads = []              # <- aquí viven las referencias
        self.init_gui()

    # (init_gui igual que con QTimer: 5 labels y un botón)

    def ejecutar_threads(self) -> None:
        for thread in self.threads:
            if thread.isRunning():
                return
        self.threads = []
```

```
31          for i in range(1, 6):
32              thread = MiThread(i, i / 10)
33              thread.senal_actualizar.connect(self.actualizar_labels)
34              self.threads.append(thread)       # ¡guardamos la referencia!
35              thread.start()
36
37      def actualizar_labels(self, indice: int, texto: str) -> None:
38          self.labels[indice].setText(texto)
```

Comparación	5 `QTimer` (Sección 3)	5 `QThread`
¿Cuántos *threads*?	Uno solo: el principal.	El principal + 5 secundarios.
¿Cómo espera?	El *event loop* llama a `enviar_dato` cada intervalo.	Un `sleep` dentro de `run`.
¿Quién toca la GUI?	El *slot* `actualizar_labels` (principal).	El mismo *slot*, vía señal encolada.
¿Concurrencia?	No.	Sí (aunque aquí no comparten estado).

¿Qué pasa si **no** guardamos la referencia?

(ejemplo de lo que NO hay que hacer)

```
1 def ejecutar_thread(self) -> None:
2     thread = MiThread()  # ¡Solo una variable local!
3     thread.senal_thread.connect(self.actualizar_label)
4     thread.start()
```

Al terminar el método, la variable local desaparece y Python puede eliminar el objeto `QThread` **mientras su *thread* sigue corriendo**:

>_ Terminal

```
QThread: Destroyed while thread '' is still running
Aborted (core dumped)
```

▲ Lo mismo pasa si reasignas el atributo

`self.mi_thread = MiThread()` mientras el *thread* anterior sigue corriendo deja al anterior sin referencias. Por eso los ejemplos revisan `isRunning()` **antes** de crear uno nuevo.

4.5 Detener un `QThread` y cerrar limpio

A diferencia de `threading.Thread`, un `QThread` **no puede ser *daemon***. Si su `run` tiene un ciclo infinito, debemos detenerlo nosotros. ¿Por qué importa?

- **Al volver a un menú.** Es muy común que en una tarea la ventana del juego se cierre para volver al inicio. Si los *threads* del juego no se detienen, siguen corriendo en el fondo, y al empezar otra partida tendrás **dos juegos a la vez** (todo va "al doble de velocidad").
- **Al cerrar el programa.** El *thread* se corta abruptamente en cualquier punto, por ejemplo a la mitad de escribir un archivo. (Con un `threading.Thread` no *daemon* pasa lo contrario: el programa no termina hasta que el *thread* termine. Pruébalo en `1_thread.py`: si cierras durante la cuenta, la terminal se queda esperando.)

La forma recomendada: una **bandera** que el ciclo de `run` revise en cada iteración, y luego **esperar** con `wait()` (el `join()` de PyQt). Aquí lo hacemos en `closeEvent`, el método que PyQt llama cuando el usuario cierra una ventana.

3-qthread/7_cierre_limpio.py

```
1 import sys
2 from time import sleep
3 from PyQt6.QtCore import pyqtSignal, QThread
4 from PyQt6.QtGui import QCloseEvent
5 from PyQt6.QtWidgets import QApplication, QWidget, QLabel
```

```python

class Contador(QThread):
    senal_actualizar = pyqtSignal(str)

    def __init__(self) -> None:
        super().__init__()
        self.activo = True                  # bandera revisada en cada iteración

    def run(self) -> None:
        segundos = 0
        while self.activo:
            # Dormimos de a poco (0.1 s) para revisar la bandera seguido.
            # Si durmiéramos 10 segundos, el cierre tardaría hasta 10 segundos
            sleep(0.1)
            segundos += 0.1
            self.senal_actualizar.emit(f"Llevo {segundos:.1f} segundos")
        print("[Contador] salí del ciclo: run terminó")

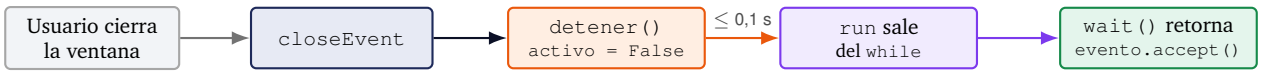
    def detener(self) -> None:
        self.activo = False


class MiVentana(QWidget):
    def __init__(self) -> None:
        super().__init__()
        self.label = QLabel("Status: iniciando", self)
        self.label.setGeometry(10, 10, 230, 30)
        self.setWindowTitle("Cierre limpio de un QThread")
        self.setGeometry(50, 50, 250, 50)
        self.contador = Contador()
        self.contador.senal_actualizar.connect(self.label.setText)
        self.contador.start()
        self.show()

    def closeEvent(self, evento: QCloseEvent) -> None:
        # Antes de cerrar, le pedimos al thread que termine y lo esperamos
        print("[Ventana] cerrando: le pido al thread que se detenga")
        self.contador.detener()
        self.contador.wait()
        print("[Ventana] el thread terminó, ahora sí cierro")
        evento.accept()
```

>_ Terminal (al cerrar la ventana)

```
[Ventana] cerrando: le pido al thread que se detenga
[Contador] salí del ciclo: run terminó
[Ventana] el thread terminó, ahora sí cierro
```



★ Detalles importantes

- El `sleep` del ciclo es **corto** (0,1 s) para revisar la bandera seguido. Si durmiera 10 s, `wait()` podría bloquear la ventana hasta 10 s.
- `QThread` trae un mecanismo equivalente a nuestra bandera: `requestInterruption()` para pedir que se detenga e `isInterruptionRequested()` para revisarlo dentro de `run`.

▲ No uses `terminate()`

Existe `terminate()`, que mata el *thread* de inmediato, pero **puede dejar *locks* tomados y datos a medio escribir**. La propia documentación de Qt lo desaconseja.

4.6 QMutex

Así como existe `QThread`, PyQt tiene su propia versión del `Lock`: `QMutex`, con `lock()` y `unlock()`. Permite que **solo un *thread* a la vez** ejecute una sección crítica. Igual que con `acquire()`/`release()`, conviene llamar

a `unlock()` dentro de un `finally`: si ocurre una excepción y el *mutex* no se libera, los demás *threads* quedan esperando **para siempre**.

En el ejemplo, 5 *threads* actualizan la ventana a distintas velocidades. El primer `for` de `run` está protegido por un `QMutex` **compartido**; el segundo no.

3-qthread/8_qmutex.py (el thread)

```python
import sys
from time import sleep
from PyQt6.QtCore import pyqtSignal, QThread, QMutex

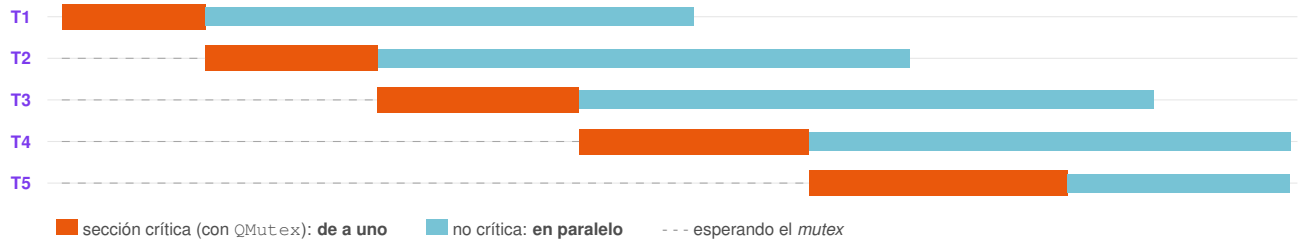

class MiThread(QThread):
    senal_actualizar = pyqtSignal(int, str)

    def __init__(self, indice: int, tiempo: float, mutex: QMutex) -> None:
        super().__init__()
        self.indice = indice
        self.tiempo = tiempo
        self.mutex = mutex                  # el MISMO mutex para todos

    def run(self) -> None:
        # Sección crítica: solo un thread a la vez. Liberamos en un finally
        self.mutex.lock()
        try:
            for i in range(10):
                sleep(self.tiempo)
                self.senal_actualizar.emit(self.indice, str(i) + " - crítico")
        finally:
            self.mutex.unlock()

        for i in range(50):
            sleep(self.tiempo)
            self.senal_actualizar.emit(self.indice, str(i) + " - no crítico")

        sleep(self.tiempo)
        self.senal_actualizar.emit(self.indice, "Status: QThread terminado")
```



★ Dos comentarios

- El `Lock` de `threading` (incluyendo `with lock:`) funciona perfectamente con `QThread`. Usa el que prefieras, pero siempre **el mismo objeto** compartido por todos.
- Si respetas la regla de oro y los *threads* solo se comunican con señales, casi todo el estado se modifica en el *thread* principal, y **rara vez vas a necesitar un *lock***.

✓ Resumen express: `QThread`

- Puedes usar `threading.Thread` (recibe la señal de la ventana) o `QThread` (define **sus propias** señales). Equivalencias: `join` → `wait`, `is_alive` → `isRunning`; no hay *daemon*.
- **Afinidad**: el objeto vive donde se creó. Solo `run` corre en el *thread* nuevo; un *slot* corre donde vive su objeto. ⇒ el trabajo pesado va en `run`.
- Crea objetos de PyQt en el *thread* principal, **nunca dentro de `run`** (allí no hay *event loop*).
- **Guarda siempre la referencia** (atributo o lista) o verás `QThread: Destroyed while thread is still running`.
- Cierre limpio: bandera + `sleep` corto + `wait()` (en `closeEvent` o `aboutToQuit`). Nunca `terminate()`.

- `QMutex` (`lock`/`unlock` en `finally`) ≡ `Lock`; con la regla de oro casi no hace falta.
- No nombres tu atributo `self.thread`: pisa el método `thread()` de PyQt.

5 Interfaces gráficas y *networking*

▲ Cómo ejecutar estos ejemplos: dos programas, dos terminales

Cada ejemplo tiene **dos programas distintos**, un servidor y un cliente, que se ejecutan por separado. Necesitas al menos dos terminales (una en la carpeta del servidor y otra en la del cliente). Ambos ejemplos usan el **mismo puerto (4444)**: antes de iniciar el servidor de un ejemplo, detén el del otro con `Ctrl+C`. **Siempre inicia primero el servidor.**

>_ Terminal 1 — servidor (siempre primero)

```
cd scripts/4-gui-y-networking/1_ejemplo_simple/servidor
python3 main.py
```

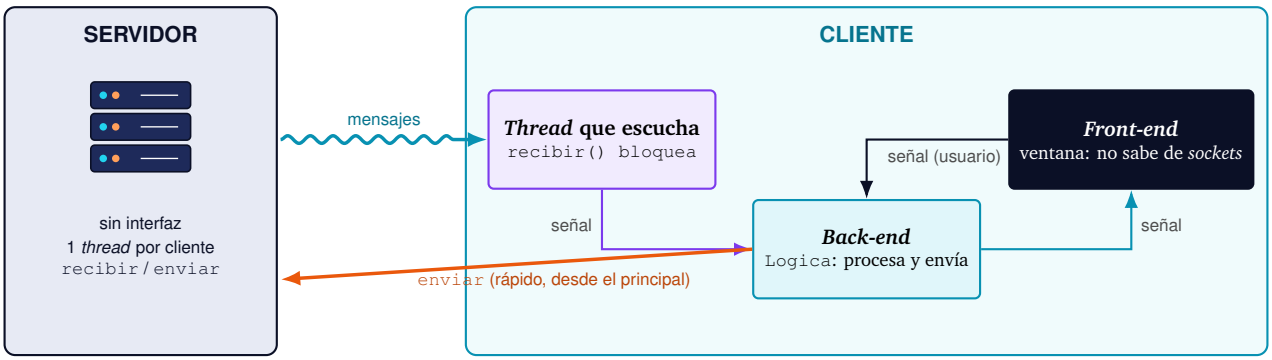
>_ Terminal 2 (y 3, 4... si quieres más clientes)

```
cd scripts/4-gui-y-networking/1_ejemplo_simple/cliente
python3 main.py
```

5.1 La arquitectura: ¿quién tiene interfaz?

En semanas anteriores vimos la arquitectura **cliente-servidor**, y la semana pasada, cómo separar un programa con interfaz en *front-end* y *back-end*. Ahora juntamos ambas cosas.

Normalmente, **el servidor no necesita interfaz gráfica**: procesa solicitudes de los clientes y nadie interactúa con él directamente. **Solo el cliente tiene interfaz**, y queda dividido en tres piezas:



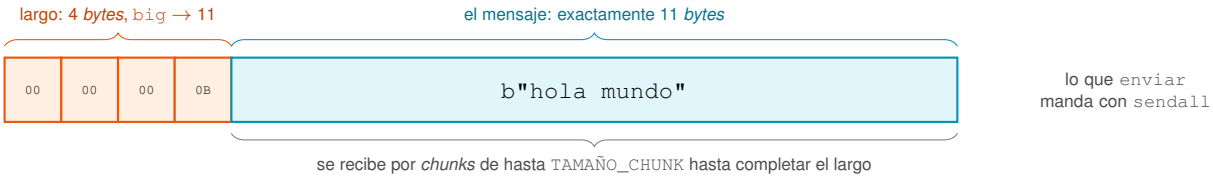
- ***Front-end***: la ventana. **No sabe nada de *sockets***: solo muestra información y emite señales cuando el usuario hace algo.
- ***Back-end***: la lógica del cliente. Envía mensajes al servidor y procesa los que llegan.
- **Un *thread* que escucha**: `recv` es **bloqueante**; si lo llamamos en el *thread* principal, la interfaz se congela hasta que llegue un mensaje (¡justo lo de la Sección 2!). Por eso la recepción va en otro *thread*, que avisa con una señal cada vez que llega algo.

★ ¿Y enviar? ¿También necesita un *thread*?

No. **Enviar es rápido** (para mensajes pequeños), así que se puede hacer directamente desde el *thread* principal. Lo que bloquea es **esperar** un mensaje que no sabemos cuándo llegará.

5.2 El protocolo

Para que los mensajes no lleguen **cortados ni pegados**, usamos el mismo protocolo de la semana de *networking*: cada mensaje se envía precedido de **4 *bytes* que indican su largo**, y se recibe por *chunks* hasta completar ese largo.



Lo dejamos en un módulo `protocolo.py`, que es **idéntico en el cliente y en el servidor**: ambos programas deben hablar el mismo protocolo.

protocolo.py (igual en cliente y servidor)

```python
import socket

TAMAÑO_CHUNK = 4096


def enviar(sock: socket.socket, mensaje: bytes) -> None:
    largo = len(mensaje).to_bytes(4, byteorder="big")
    sock.sendall(largo + mensaje)


def recibir(sock: socket.socket) -> bytes:
    largo = int.from_bytes(recibir_bytes(sock, 4), byteorder="big")
    return recibir_bytes(sock, largo)


def recibir_bytes(sock: socket.socket, cantidad: int) -> bytes:
    datos = bytearray()
    while len(datos) < cantidad:
        chunk = sock.recv(min(TAMAÑO_CHUNK, cantidad - len(datos)))
        # recv retorna b"" cuando la otra parte cerró la conexión
        if not chunk:
            raise ConnectionError("La conexión se cerró")
        datos.extend(chunk)
    return bytes(datos)
```

La pieza clave para detectar desconexiones

`recibir_bytes` lanza **`ConnectionError`** cuando `recv` retorna `b""`, lo que ocurre cuando **la otra parte cerró la conexión**. Así, quien llame a `recibir` puede detectar una desconexión con un simple `try/except`. Además, `min(TAMAÑO_CHUNK, cantidad - len(datos))` evita leer *bytes* del **siguiente** mensaje.

5.3 Ejemplo simple

Una ventana desde la cual se le puede pedir al servidor **un saludo o un color**. El servidor responde con uno al azar, y la respuesta se muestra en la ventana. La estructura de archivos es:

```
1_ejemplo_simple/
├── servidor/
│   ├── main.py
│   ├── constantes.py
│   └── protocolo.py
└── cliente/
    ├── main.py
    ├── constantes.py
    ├── protocolo.py
    ├── backend/
    │   └── logica.py
    └── frontend/
        └── ventana.py
```

Servidor: un *thread* por cliente, sin interfaz.

`main.py`: crea *back-end* y *front-end* y **conecta sus señales**.

`protocolo.py`: **idéntico** en ambos lados.

`logica.py`: `EscucharQThread` + `Logica`.

`ventana.py`: botones y *labels*; nada nuevo de esta semana.

5.3.1 El servidor

Igual a los de la semana de *networking*: un *thread* por cliente, que recibe un comando y responde.

1_ejemplo_simple/servidor/main.py (fragmento)

```python
import socket
from random import choice
from threading import Thread
from constantes import HOST_DEFECTO, PORT_DEFECTO, SALUDOS, COLORES
from protocolo import enviar, recibir


# Función encargada de escuchar a 1 cliente
def escuchar_cliente(socket_cliente: socket.socket, direccion: tuple) -> None:
    try:
        while True:
            mensaje = recibir(socket_cliente).decode("utf-8")
            print(f"[{direccion}] pidió: {mensaje}")
            if mensaje == "palabra":
                respuesta = choice(SALUDOS)
            else:  # mensaje == "color"
                respuesta = choice(COLORES)
            enviar(socket_cliente, respuesta.encode("utf-8"))

    except ConnectionError:
        # El cliente se desconectó: termina el thread que lo atendía
        print(f"[{direccion}] se desconectó")
    finally:
        socket_cliente.close()
```

5.3.2 El *thread* que escucha

En el cliente, el *back-end* está en `backend/logica.py`. Primero, un `QThread` cuya **única tarea es escuchar**: en un ciclo infinito llama a `recibir` (que bloquea hasta que llegue algo) y emite una señal con cada mensaje. Si la conexión se pierde, emite otra señal y termina.

1_ejemplo_simple/cliente/backend/logica.py (parte 1)

```python
import socket
from PyQt6.QtCore import QObject, QThread, pyqtSignal
from protocolo import enviar, recibir


# recv es bloqueante: no puede ejecutarse en el thread principal
class EscucharQThread(QThread):
    senal_mensaje = pyqtSignal(str)       # entrega cada mensaje a la Logica
    senal_desconectado = pyqtSignal()     # avisa que se perdió la conexión

    def __init__(self, socket_cliente: socket.socket) -> None:
        super().__init__()
        self.socket_cliente = socket_cliente

    def run(self) -> None:
        print("Levantando el thread que escucha al servidor...")
        try:
            while True:
                mensaje = recibir(self.socket_cliente).decode("utf-8")
                self.senal_mensaje.emit(mensaje)
        except (ConnectionError, OSError):
            # El servidor se cayó, o nosotros cerramos el socket
            print("Dejé de escuchar al servidor")
            self.senal_desconectado.emit()
```

5.3.3 La lógica del cliente

La clase `Logica` crea el *socket*, se conecta, inicia el *thread* que escucha (**guardando su referencia**) y procesa los mensajes. `procesar_mensaje` está conectado a la señal del *thread*, así que **se ejecuta en el *thread* principal** y puede emitir señales al *front-end* sin problemas.

1_ejemplo_simple/cliente/backend/logica.py (parte 2)

```python
class Logica(QObject):
    senal_actualizar_etiqueta = pyqtSignal(str)
    senal_desconectado = pyqtSignal()

    def __init__(self, host: str, port: int) -> None:
        super().__init__()
        self.host = host
        self.port = port
        self.socket_cliente = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.thread_escuchar = None

    def conectar_a_servidor(self) -> bool:
        try:
            self.socket_cliente.connect((self.host, self.port))
        except ConnectionError:
            print("No me pude conectar al servidor.")
            return False
        print("Cliente conectado exitosamente al servidor.")
        # Creamos el thread que escucha y guardamos su referencia
        self.thread_escuchar = EscucharQThread(self.socket_cliente)
        self.thread_escuchar.senal_mensaje.connect(self.procesar_mensaje)
        # Una señal se puede conectar a OTRA señal: la "reenvía"
        self.thread_escuchar.senal_desconectado.connect(self.senal_desconectado)
        self.thread_escuchar.start()
        return True

    # Para este ejemplo, solo lo mandamos al frontend
    def procesar_mensaje(self, mensaje: str) -> None:
        self.senal_actualizar_etiqueta.emit(mensaje)

    def mandar_comando(self, comando: str) -> None:
        try:
            enviar(self.socket_cliente, comando.encode("utf-8"))
        except (ConnectionError, OSError):
            self.senal_desconectado.emit()

    def cerrar(self) -> None:
        # shutdown despierta al recv que está bloqueado en el otro thread
        # (le hace retornar b""), así el thread termina por sí solo
        try:
            self.socket_cliente.shutdown(socket.SHUT_RDWR)
        except OSError:
            pass  # El socket ya estaba desconectado
        self.socket_cliente.close()
        if self.thread_escuchar is not None:
            self.thread_escuchar.wait()
```

★ Conectar una señal a otra señal

`self.thread_escuchar.senal_desconectado.connect(self.senal_desconectado)` conecta una señal **a otra señal**: cuando el *thread* emite la suya, `Logica` emite automáticamente `senal_desconectado`, que el `main.py` conecta con la ventana. Es una forma limpia de "reenviar" un aviso sin escribir un método intermedio.

5.3.4 Conectando todo en `main.py`

`main.py` crea el *back-end* y el *front-end* y **conecta sus señales**. La ventana no tiene nada nuevo: emite `senal_mandar_comando` al presionar un botón y tiene los métodos `actualizar_etiqueta` y `mostrar_desconexion`.

1_ejemplo_simple/cliente/main.py (fragmento)

```python
    app = QApplication([])
    backend = Logica(HOST_DEFECTO, PORT_DEFECTO)
    frontend = MainWindow()

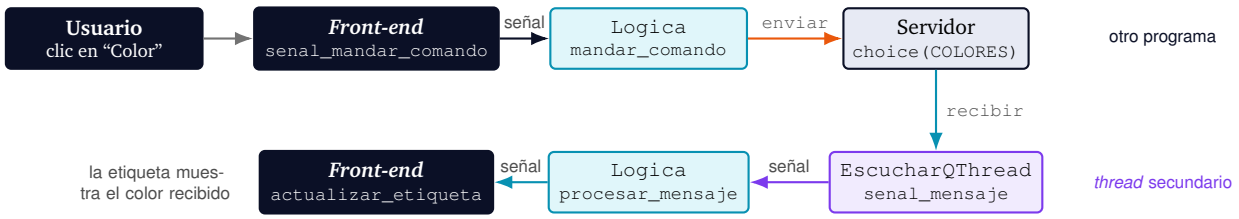
    # Conectamos señales
    backend.senal_actualizar_etiqueta.connect(frontend.actualizar_etiqueta)
    backend.senal_desconectado.connect(frontend.mostrar_desconexion)
    frontend.senal_mandar_comando.connect(backend.mandar_comando)
```

```python
    # Justo antes de que termine la aplicación, cerramos la conexión
    app.aboutToQuit.connect(backend.cerrar)

    if not backend.conectar_a_servidor():
        sys.exit(1)

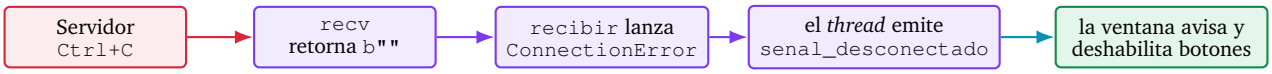
    frontend.show()
    sys.exit(app.exec())
```

`app.aboutToQuit` es una señal que se emite **justo antes de que termine la aplicación**. La usamos para cerrar la conexión y esperar al *thread* que escucha, igual que en el cierre limpio de la Sección 4.5.



5.3.5 ¿Qué pasa si el servidor se cae?

Pruébalo: con un cliente abierto, detén el servidor con `Ctrl+C`.



La ventana muestra "Se perdió la conexión con el servidor" y deshabilita sus botones, **en vez de quedarse pegada o caerse**. Y el *thread* que escucha termina solo (sale del `while True` por la excepción).

El truco de `shutdown` en `cerrar`

Cuando **nosotros** cerramos el cliente, el *thread* que escucha está **bloqueado en `recv`**. Si solo llamáramos a `wait()`, esperaríamos para siempre. `shutdown(socket.SHUT_RDWR)` hace que ese `recv` **retorne de inmediato**; `recibir` lanza la excepción, el *thread* sale de su ciclo y `wait()` retorna. Es exactamente la "bandera" del cierre limpio, pero para un *thread* que está esperando la red.

★ `OSError: [Errno 98] Address already in use`

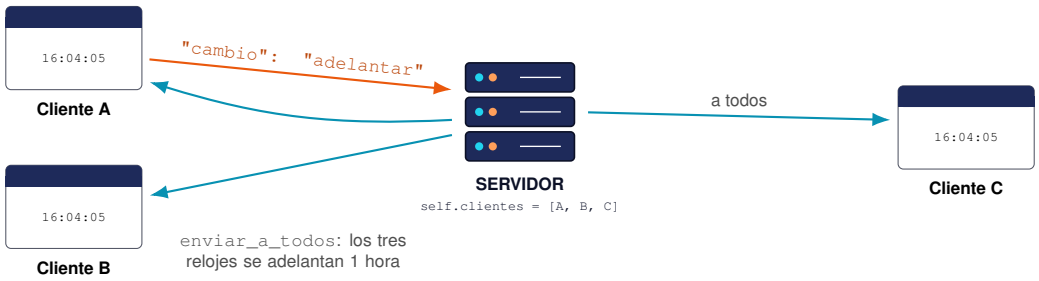
Si reinicias el servidor apenas lo detuviste, el sistema operativo puede mantener el puerto reservado un rato. Por eso ambos servidores llaman a `setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)` **antes del `bind`**, lo que permite reutilizarlo de inmediato. **Ojo**: `SO_REUSEADDR` no sirve si **otro programa está escuchando** en ese puerto (por ejemplo, el servidor del otro ejemplo, que usa el mismo). En ese caso, cierra ese otro servidor.

5.4 Ejemplo avanzado: interacción entre clientes

Ahora hay interacción **entre clientes**, con el servidor como intermediario. Al conectarse, cada cliente pide al servidor la **hora de Chile** y la muestra como un reloj (por simplicidad, el servidor usa una diferencia fija con UTC: la de Chile en horario de verano, UTC−3). También puede pedir la hora de Japón, o pedir **adelantar o retrasar el reloj en 1 hora**: estas dos últimas acciones afectan **a todos los clientes conectados**. Pruébalo con al menos dos clientes.

Esta vez los mensajes son **diccionarios serializados con JSON**:

Envía	Mensaje	Respuesta del servidor
Cliente	`{"comando": "hora", "pais": "chile"}`	`{"tipo": "hora", "hora": "2026-11-02T15:04:05"}`, **solo a ese cliente**
Cliente	`{"comando": "cambio", "accion": "adelantar"}`	`{"tipo": "cambio", "accion": "adelantar"}`, a **todos** los clientes



5.4.1 El servidor

El servidor guarda en una **lista** los *sockets* de todos los clientes conectados, para poder enviarles mensajes a todos. Esa lista es **compartida por todos los *threads*** que atienden clientes, así que la protegemos con un `Lock`. Cuando un cliente se desconecta, lo sacamos de la lista en el `finally`.

2_ejemplo_avanzado/servidor/main.py (fragmento)

```python
class Servidor:
    def __init__(self, host: str, port: int) -> None:
        # Lista compartida por todos los threads: la protegemos con un Lock
        self.clientes = []
        self.lock_clientes = Lock()
        # Dos threads pueden enviarle al mismo cliente a la vez (la respuesta
        # a "hora" y un envío a todos). Este lock evita que los bytes de
        # ambos mensajes se mezclen en el socket y rompan el protocolo
        self.lock_envio = Lock()

        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        # Permite reutilizar el puerto apenas se cierra el servidor
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.sock.bind((host, port))
        self.sock.listen()

    def escuchar_conexiones(self) -> None:
        print("Empezando a escuchar clientes....")
        while True:
            socket_cliente, direccion = self.sock.accept()
            print("Conexión aceptada desde", direccion)
            with self.lock_clientes:
                self.clientes.append(socket_cliente)
            thread = Thread(
                target=self.escuchar_cliente,
                args=(socket_cliente, direccion),
                daemon=True,
            )
            thread.start()

    def escuchar_cliente(self, socket_cliente: socket.socket, direccion: tuple) -> None:
        try:
            while True:
                mensaje = json.loads(recibir(socket_cliente).decode("utf-8"))
                print(f"[{direccion}] mandó un mensaje: {mensaje}")
                self.procesar_mensaje(mensaje, socket_cliente)
        except ConnectionError:
            print(f"[{direccion}] se desconectó")
        finally:
            # Sacamos al cliente de la lista, para no enviarle más mensajes
            with self.lock_clientes:
                self.clientes.remove(socket_cliente)
            socket_cliente.close()

    def procesar_mensaje(self, mensaje: dict, socket_cliente: socket.socket) -> None:
        if mensaje["comando"] == "cambio":
            # "adelantar" o "retrasar" afecta a TODOS los clientes conectados
            respuesta = {"tipo": "cambio", "accion": mensaje["accion"]}
            self.enviar_a_todos(respuesta)
        elif mensaje["comando"] == "hora":
            # La hora de un país se responde solo a quien la pidió
            horas = DIFERENCIAS_HORARIAS[mensaje["pais"]]
            ahora_utc = datetime.datetime.now(datetime.UTC)
            hora_pais = ahora_utc + datetime.timedelta(hours=horas)
```

```
            respuesta = {"tipo": "hora", "hora": hora_pais.replace(tzinfo=None).isoformat()}
            self.enviar(socket_cliente, respuesta)

    def enviar(self, socket_cliente: socket.socket, mensaje: dict) -> None:
        # Cada mensaje se envía completo antes de que empiece el siguiente
        with self.lock_envio:
            enviar(socket_cliente, json.dumps(mensaje).encode("utf-8"))

    def enviar_a_todos(self, mensaje: dict) -> None:
        # Copiamos la lista dentro de lock_clientes, y enviamos fuera de él
        with self.lock_clientes:
            clientes = list(self.clientes)
        for socket_cliente in clientes:
            try:
                self.enviar(socket_cliente, mensaje)
            except OSError:
                # Ese cliente se desconectó: su propio thread lo sacará de la lista
                pass
```

Lock	Protege	Detalle
`lock_clientes`	La lista `self.clientes` (agregar, quitar, copiar).	En `enviar_a_todos` se **copia la lista dentro** del *lock* y se **envía fuera**: así no lo tenemos tomado mientras esperamos a la red.
`lock_envio`	Que un mensaje se envíe **completo** antes que otro.	Sin él, la respuesta a "hora" y un envío a todos podrían **mezclar sus *bytes*** en el mismo *socket* y romper el protocolo. Solo hace esperar a otros envíos.

5.4.2 El cliente: un *thread* para escuchar y un `QTimer` para el reloj

El *back-end* del cliente usa **las dos herramientas de la semana**, cada una para lo que corresponde:

Recibir mensajes bloquea ⇒ `EscucharQThread`
el mismo del ejemplo simple, ahora con mensajes JSON

El reloj avanza 1 s cada 1 s ⇒ `QTimer`
tarea periódica y rápida: sin *lock* para `hora_actual`

2_ejemplo_avanzado/cliente/backend/logica.py (fragmento)

```
class Logica(QObject):
    senal_actualizar_etiqueta = pyqtSignal(str)
    senal_desconectado = pyqtSignal()

    def __init__(self, host: str, port: int) -> None:
        super().__init__()
        self.host = host
        self.port = port
        self.socket_cliente = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.thread_escuchar = None
        # Tarea periódica y corta: QTimer. Todo ocurre en el thread
        # principal, por lo que self.hora_actual no necesita un lock
        self.hora_actual = None
        self.timer_reloj = QTimer(self)
        self.timer_reloj.setInterval(1000)
        self.timer_reloj.timeout.connect(self.avanzar_reloj)

    def conectar_a_servidor(self) -> bool:
        try:
            self.socket_cliente.connect((self.host, self.port))
        except ConnectionError:
            print("No me pude conectar al servidor.")
            return False
        print("Cliente conectado exitosamente al servidor.")
        # Recibir mensajes es bloqueante, así que usamos un QThread
        self.thread_escuchar = EscucharQThread(self.socket_cliente)
        self.thread_escuchar.senal_mensaje.connect(self.procesar_mensaje)
        self.thread_escuchar.senal_desconectado.connect(self.desconectado)
        self.thread_escuchar.start()
        self.pedir_hora("chile")
        return True
```

```python

    def procesar_mensaje(self, mensaje: dict) -> None:
        if mensaje["tipo"] == "hora":
            self.hora_actual = datetime.datetime.fromisoformat(mensaje["hora"])
            if not self.timer_reloj.isActive():
                self.timer_reloj.start()
        elif mensaje["tipo"] == "cambio" and self.hora_actual is not None:
            una_hora = datetime.timedelta(hours=1)
            if mensaje["accion"] == "adelantar":
                self.hora_actual += una_hora
            else:  # mensaje["accion"] == "retrasar"
                self.hora_actual -= una_hora
        self.actualizar_etiqueta()

    def avanzar_reloj(self) -> None:
        self.hora_actual += datetime.timedelta(seconds=1)
        self.actualizar_etiqueta()

    def actualizar_etiqueta(self) -> None:
        if self.hora_actual is not None:
            self.senal_actualizar_etiqueta.emit(self.hora_actual.strftime("%Y-%m-%d %H:%M:%S"))

    def desconectado(self) -> None:
        self.timer_reloj.stop()
        self.senal_desconectado.emit()
```

¿Por qué `hora_actual` no necesita lock*?* La modifican dos métodos: `procesar_mensaje` (conectado a la señal del *thread*, pero ejecutado en el principal por la conexión encolada) y `avanzar_reloj` (llamado por el `QTimer`, también en el principal). **Ambos corren en el mismo *thread***, uno después del otro: nunca a la vez.

★ Acotación: ¿y con `threading`?

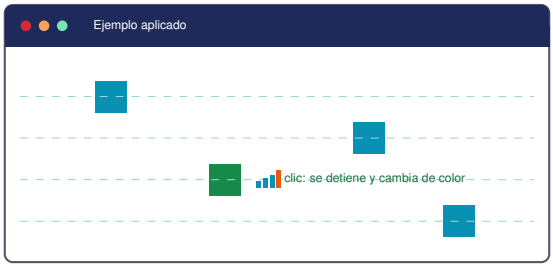
El *thread* que escucha también se puede implementar con `threading.Thread`, igual que en el primer ejemplo de la Sección 4: recibe una señal definida en `Logica` y la emite con cada mensaje. El resultado es el mismo, porque la regla de oro se cumple igual. Gracias a la separación *front-end*/*back-end*, ese cambio **solo afectaría a `backend/logica.py`**: la ventana ni se enteraría.

✓ Resumen express: GUI y *networking*

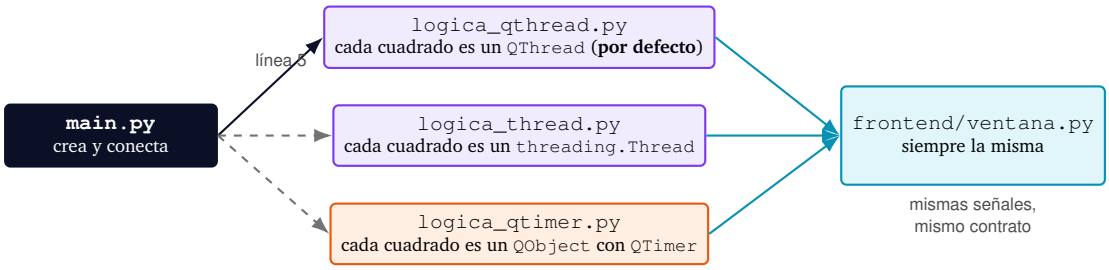
- Solo el **cliente** tiene interfaz: *front-end* (ventana, sin *sockets*) + *back-end* (`Logica`) + un ***thread* que escucha**.
- `recv` bloquea ⇒ va en un *thread* que emite una señal por mensaje. **Enviar** es rápido ⇒ desde el principal.
- Protocolo: 4 *bytes* de largo + mensaje, recibido por *chunks*. `recv` retorna `b""` ⇒ `ConnectionError` ⇒ `senal_desconectado`.
- Cierre: `app.aboutToQuit.connect(backend.cerrar)`; `shutdown` despierta al `recv` bloqueado y luego `wait()`.
- Servidor con varios clientes: lista de *sockets* con `Lock`; copiar dentro del *lock*, enviar fuera; un `lock_envio` para no mezclar *bytes*.
- `SO_REUSEADDR` antes del `bind` evita `Address already in use` al reiniciar rápido.

6 Ejemplo aplicado: un *front-end*, tres *back-ends*

Para cerrar, juntemos todo: modularización, *front-end* y *back-end*, señales, *threads* y `QTimer`. Varios cuadrados van de un extremo a otro de la ventana, cada uno a distinta velocidad. **Al hacer clic en un cuadrado, cambia de color y se detiene.** (Mini desafío: intenta detenerlos todos.)



Lo interesante es que **el mismo juego está implementado con tres *back-ends* distintos**, que se intercambian **sin tocar el *front-end***:



Archivos generales

- `main.py`: crea *front-end* y *back-end* y conecta sus señales.
- `parametros_general.py`: constantes de ambos.

frontend/

- `ventana.py`: la ventana y todo lo visual.
- `parametros_frontend.py`: constantes solo del *front-end*.

backend/

- `logica_qthread.py`: un `QThread` por cuadrado (**por defecto**).
- `logica_thread.py`: idéntico, salvo la herencia y dos detalles: sus *threads* son *daemon*, y al cerrar se esperan con `join()` en vez de `wait()`.
- `logica_qtimer.py`: un `QObject` con su propio `QTimer` por cuadrado.
- `parametros_backend.py`: constantes solo del *back-end*.

Para cambiar de *back-end* basta con cambiar **la línea 5** de `main.py`:

5-ejemplo-aplicado/main.py (inicio)

```python
from PyQt6.QtWidgets import QApplication
from frontend.ventana import Ventana
import sys
import parametros_general as p
from backend.logica_qthread import Juego

# El mismo juego está implementado con 3 back-ends distintos. Para probar
# otro, cambia la línea 5 por una de las siguientes:
# from backend.logica_thread import Juego    # threading.Thread
# from backend.logica_qtimer import Juego    # QTimer
```

6.1 Comparando los tres *back-ends*

La diferencia está en **cómo se mueve cada cuadrado** (`Icono`). Con `QThread` (y lo mismo con `Thread`), cada cuadrado tiene **su propio ciclo con un `sleep`** en `run`, y se detiene con una bandera:

backend/logica_qthread.py (run de cada Icono)

```python
    def run(self) -> None:
        # Mientras pueda correr
        while self.correr:
            # Me muevo y luego espero un poco de tiempo.
            self.mover()
            # sleep recibe segundos: dividimos p.TIEMPO_MOVIMIENTO (ms) por 1000
            time.sleep(p.TIEMPO_MOVIMIENTO / 1000)
```

Con `QTimer`, en cambio, **no hay ciclo ni `sleep`**: el *timer* llama a `mover` cada `TIEMPO_MOVIMIENTO` milisegundos, y se detiene con `stop()`:

backend/logica_qtimer.py (__init__ de cada Icono)

```python
        # QTimer encargado de ejecutar "self.mover" cada p.TIEMPO_MOVIMIENTO
        self.timer_movimiento = QTimer(self)
        self.timer_movimiento.timeout.connect(self.mover)
        self.timer_movimiento.setInterval(p.TIEMPO_MOVIMIENTO)
```

	Thread / QThread	QTimer
¿Dónde se ejecuta `mover`?	En el *thread* de cada cuadrado	En el *thread* principal
¿Cómo se detiene un cuadrado?	Bandera `correr = False`	`timer.stop()`
¿Riesgo de concurrencia?	✗ Sí: cuidar el estado compartido	✓ No
¿Hay que detenerlo al cerrar?	Sí: bandera + `wait()`/`join()`	Basta con `stop()`

¿Cuál elegir?

Los tres funcionan, pero según la tabla de la Sección 2, mover un cuadrado unos pocos píxeles es una **tarea periódica y rápida**: `QTimer` es la opción **más simple**. Y si los cuadrados tuvieran que **interactuar entre sí** (chocar), un **solo `QTimer` para todo el juego**, como en el *loop* de juego de la Sección 3, sería todavía más cómodo.

★ Cierre ordenado

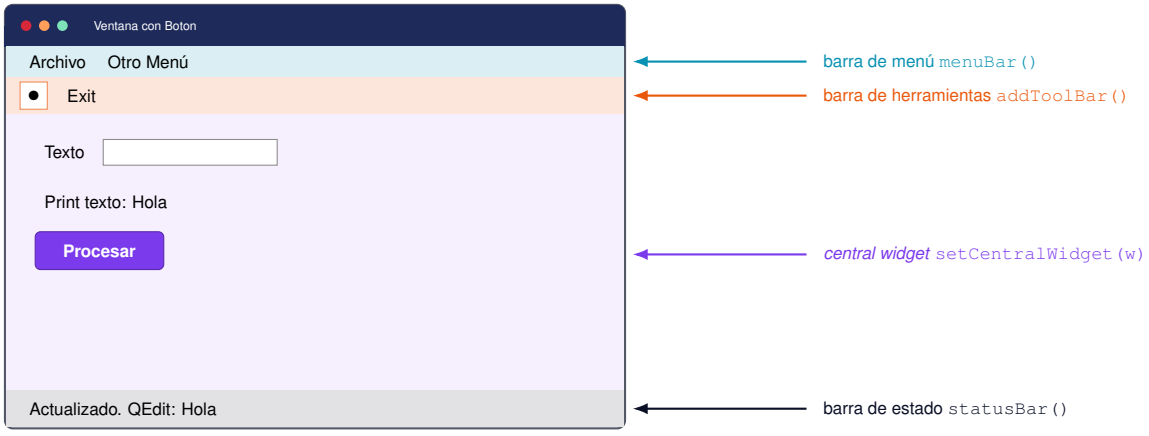
`main.py` conecta `app.aboutToQuit` con el método `detener` del *back-end*, para que los *threads* o *timers* terminen ordenadamente al cerrar (Sección 4.5).

7 Recetario: *tips* útiles

Herramientas que no son el centro de la semana, pero te servirán. Como estos ejemplos usan imágenes con *paths* relativos, la terminal **debe** estar en `scripts/5-recetario`.

7.1 QMainWindow

Las ventanas que hemos creado con `QWidget` son ventanas simples. PyQt ofrece una más completa, `QMainWindow`, con el **esqueleto clásico** de una aplicación:



- La **barra de menú** (`menuBar()`) agrupa los comandos en menús.
- La **barra de herramientas** (`addToolBar()`) da acceso rápido a los más usados.
- La **barra de estado** (`statusBar()`) muestra información del estado de la aplicación.
- El ***central widget*** es el cuerpo de la ventana; puede ser **cualquier** *widget*, incluso una de las ventanas de los ejemplos anteriores (`setCentralWidget(widget)`).

5-recetario/1_mainwindow.py

```python
import os
import sys
from PyQt6.QtCore import pyqtSignal
from PyQt6.QtGui import QIcon, QAction
from PyQt6.QtWidgets import QApplication, QMainWindow, QWidget
from PyQt6.QtWidgets import QPushButton, QLabel, QLineEdit


class MiVentana(QWidget):
    def __init__(self, status_bar_signal: pyqtSignal) -> None:
        super().__init__()
        self.status_bar_signal = status_bar_signal
        self.init_gui()

    def init_gui(self) -> None:
        self.label_text = QLabel("Texto", self)
        self.print_label = QLabel("Print texto:", self)
        self.line_edit = QLineEdit("", self)
        self.boton = QPushButton("&Procesar", self)
        self.boton.resize(self.boton.sizeHint())
        self.boton.clicked.connect(self.boton_callback)
        self.label_text.move(20, 15)
        self.line_edit.setGeometry(55, 15, 100, 20)
        self.print_label.move(20, 50)
        self.boton.move(20, 80)

    def boton_callback(self) -> None:
        # Actualiza su label y avisa a la barra de estado con una señal
        self.print_label.setText(f"Print texto: {self.line_edit.text()}")
        self.print_label.resize(self.print_label.sizeHint())
        self.status_bar_signal.emit(f"QEdit: {self.line_edit.text()}")


class MainWindow(QMainWindow):
    # Comunica la barra de estado con el resto de los widgets
    onchange_status_bar = pyqtSignal(str)

    def __init__(self) -> None:
        super().__init__()
        self.setWindowTitle("Ventana con Boton")
        self.setGeometry(200, 100, 300, 250)

        # Acciones: texto, tip para la barra de estado y slot
        ver_status = QAction(QIcon(None), "&Cambiar Status", self)
        ver_status.setStatusTip("Este es un ítem de prueba")
        ver_status.triggered.connect(self.cambiar_status_bar)

        limpiar_status = QAction(QIcon(None), "&Limpiar Status", self)
        limpiar_status.setStatusTip("Esta acción limpia la barra de estado")
        limpiar_status.triggered.connect(self.limpiar_status_bar)

        buscar = QAction(QIcon(os.path.join("img", "search_icon.png")), "&Search", self)
        buscar.setStatusTip("Un ícono de búsqueda")

        salir = QAction(QIcon(None), "&Exit", self)
        salir.setShortcut("Ctrl+Q")
        salir.setStatusTip("Salir de la aplicación")
        salir.triggered.connect(QApplication.quit)

        # Barra de menú
        menubar = self.menuBar()
        archivo_menu = menubar.addMenu("Archivo")
        archivo_menu.addAction(ver_status)
        archivo_menu.addAction(salir)
        otro_menu = menubar.addMenu("Otro Menú")
        otro_menu.addAction(limpiar_status)

        # Barra de herramientas
        toolbar = self.addToolBar("Toolbar")
        toolbar.addAction(buscar)
        toolbar.addAction(salir)

```

```python
        # Barra de estado
        self.statusBar().showMessage("Listo")
        self.onchange_status_bar.connect(self.actualizar_status_bar)

        # Widget central: le pasamos la señal para que hable con la barra de estado
        self.form = MiVentana(self.onchange_status_bar)
        self.setCentralWidget(self.form)

    def cambiar_status_bar(self) -> None:
        self.statusBar().showMessage("Cambié el Status")

    def limpiar_status_bar(self) -> None:
        self.statusBar().showMessage("Status limpio.")

    def actualizar_status_bar(self, msg: str) -> None:
        self.statusBar().showMessage(f"Actualizado. {msg}")
```

★ Piezas nuevas de este ejemplo

- `QAction` (en `PyQt6.QtGui`) representa un **comando** reutilizable: el mismo `salir` está en el menú y en la barra de herramientas. Su señal es `triggered`, y `setShortcut("Ctrl+Q")` le da un atajo.
- `setStatusTip` muestra un texto en la barra de estado al pasar el *mouse* sobre la acción.
- El `&` en `"&Procesar"` define la tecla de acceso rápido (`Alt+P`).

▲ En macOS

La barra de menú **no aparece dentro de la ventana**, sino en la barra superior de la pantalla, junto al logo de Apple. No es un error de tu código.

7.2 `isAutoRepeat` en `keyPressEvent`

Si mantenemos una tecla presionada, por defecto `keyPressEvent` se llama **muchas veces**, como cuando mantienes una tecla en un editor de texto. A veces es lo que queremos (mover un personaje), pero otras queremos reaccionar **solo la primera vez** (pausar el juego).

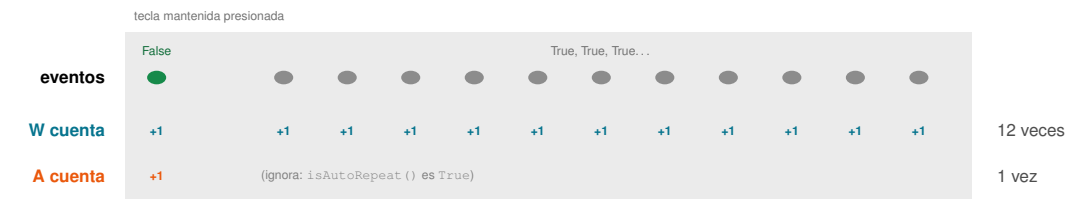
■ `evento.isAutoRepeat()`

Retorna `False` cuando la tecla **se acaba de presionar**, y `True` cuando el evento se repite por mantenerla presionada.

5-recetario/2_autorepeat.py (fragmento)

```python
    def keyPressEvent(self, evento: QKeyEvent) -> None:
        if evento.key() == Qt.Key.Key_W:
            self.contador_w += 1
            self.label_w_contador.setText(f"Presionada {self.contador_w} veces")

        if evento.key() == Qt.Key.Key_A and not evento.isAutoRepeat():
            self.contador_a += 1
            self.label_a_contador.setText(f"Presionada {self.contador_a} veces")
```



■ Qué verás

Mantén `W` y `A` presionadas: el contador de `W` sube sin parar, el de `A` sube **una sola vez** por cada pulsación. Es lo que usamos para la pausa del *loop* de juego.

7.3 Versiones de PyQt

Todo este material usa **PyQt6**. En internet encontrarás mucho código de PyQt5, que es parecido pero no idéntico:

PyQt5 (no usar)	PyQt6 (el del curso)
`app.exec_()`	`app.exec()`
`Qt.LeftButton, Qt.Key_W`	`Qt.MouseButton.LeftButton, Qt.Key.Key_W`
`QAction en PyQt5.QtWidgets`	`QAction en PyQt6.QtGui`
`QMediaPlayer + QMediaContent + setMedia`	`QMediaPlayer + QAudioOutput + setSource`

Si copias código de otra fuente, **revisa que corresponda a PyQt6**.

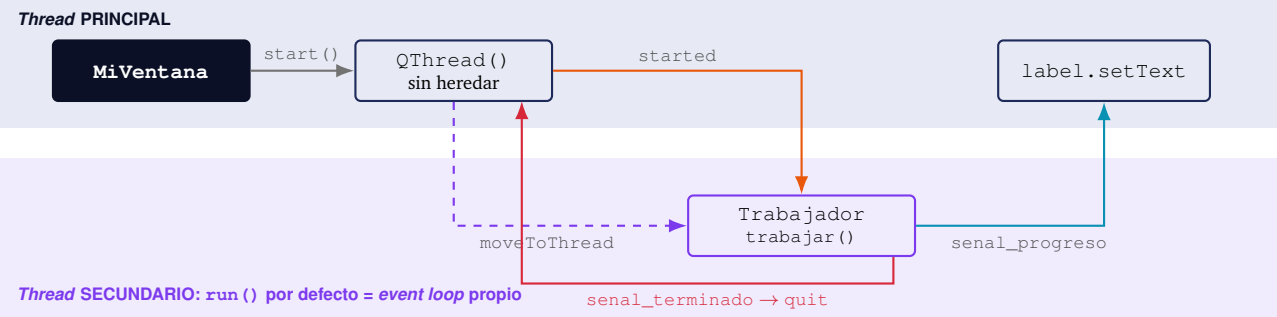
8 Bonus: patrón *worker* y sonidos

★ Contenido *bonus*

Esta sección no es parte de los contenidos de la semana, pero te puede servir para profundizar o para tu tarea.

8.1 El patrón *worker*: `moveToThread`

En la Sección 4 usamos el patrón más directo: heredar de `QThread` y escribir el trabajo en `run`. La documentación de Qt recomienda además otro patrón, que **evita la confusión de afinidad**: crear un `QObject` común (el *worker*) y **moverlo** a un `QThread` con `moveToThread`. Así, **todos** los métodos del *worker* conectados a señales se ejecutan en el *thread* secundario.



bonus-worker-y-sonidos/1_move_to_thread.py

```python
class Trabajador(QObject):
    """QObject común (NO hereda de QThread). Sus métodos se ejecutarán
    en el thread al que lo movamos con moveToThread."""

    senal_progreso = pyqtSignal(str)
    senal_terminado = pyqtSignal()

    def trabajar(self) -> None:
        print(f"[trabajar] me ejecuto en el {donde_estoy()}")    # SECUNDARIO
        for i in range(1, 6):
            sleep(0.5)
            self.senal_progreso.emit(f"Paso {i} de 5")
        self.senal_progreso.emit("Status: trabajo terminado")
        self.senal_terminado.emit()


class MiVentana(QWidget):
    def __init__(self) -> None:
        super().__init__()
        self.label = QLabel("Status: esperando", self)
        self.boton = QPushButton("Trabajar", self)
        self.label.setGeometry(10, 10, 230, 30)
        self.boton.setGeometry(10, 50, 230, 30)
```

```
        self.boton.clicked.connect(self.iniciar)
        self.setWindowTitle("Patrón worker + moveToThread")
        self.setGeometry(50, 50, 250, 90)

        # 1. Un QThread "vacío": su run por defecto solo corre un event loop
        self.mi_thread = QThread()
        # 2. Creamos el trabajador y lo movemos a ese thread
        self.trabajador = Trabajador()
        self.trabajador.moveToThread(self.mi_thread)
        # 3. Conectamos señales
        self.trabajador.senal_progreso.connect(self.label.setText)
        self.trabajador.senal_terminado.connect(self.mi_thread.quit)
        self.mi_thread.started.connect(self.trabajador.trabajar)
        self.show()

    def iniciar(self) -> None:
        if not self.mi_thread.isRunning():
            self.mi_thread.start()
```

¿Cómo funciona? El `QThread` **no se hereda**: su `run` por defecto ejecuta un ***event loop* para ese *thread***, y es ese *event loop* el que ejecuta `trabajar` cuando se emite `started` (conexión encolada hacia el *thread* donde ahora vive el *worker*). Al terminar, el *worker* emite `senal_terminado`, conectada a `self.mi_thread.quit`, que detiene ese *event loop*. ¡Compáralo con el problema de T1/T2: aquí el *thread* secundario **sí** tiene *event loop*!

8.2 Sonidos en PyQt6

PyQt6 ofrece dos clases para reproducir audio, en `PyQt6.QtMultimedia`:

	QSoundEffect	QMediaPlayer
Formatos	Solo `.wav`	Muchos, incluyendo `.mp3`
Configuración	`setSource(url)` y listo	Necesita además un `QAudioOutput`
Uso típico	Efectos cortos que deben sonar **sin retraso**	Música y audios largos o comprimidos

Ambas reciben la ubicación del archivo como un `QUrl`, creado con `QUrl.fromLocalFile(path)`. Para `QMediaPlayer` construimos un *path* **absoluto** con `os.path.abspath`.

bonus-worker-y-sonidos/2_sonidos.py (fragmento)

```
        # Opción MP3: QMediaPlayer (reproduce muchos formatos, como MP3)
        # 1. Para que QMediaPlayer reproduzca audio, se debe agregar un QAudioOutput
        # 2. Usamos el path absoluto (os.path.abspath)
        # 3. setSource indica el path al audio a escuchar
        self.player = QMediaPlayer(self)
        self.audio_output = QAudioOutput(self)
        self.player.setAudioOutput(self.audio_output)
        path = os.path.abspath(join("sounds", "waku-waku.mp3"))
        self.player.setSource(QUrl.fromLocalFile(path))

        # Opción WAV: QSoundEffect (solo reproduce archivos WAV)
        self.media_player_wav = QSoundEffect(self)
        self.media_player_wav.setVolume(0.1)  # Opcional
        # Como es música de fondo, la repetimos indefinidamente
        self.media_player_wav.setLoopCount(QSoundEffect.Loop.Infinite.value)
        file_url = QUrl.fromLocalFile(join("sounds", "see-you-again.wav"))
        self.media_player_wav.setSource(file_url)
        self.show()

    def empezar_sonido_sorpresa(self) -> None:
        self.player.play()

    def empezar_musica_fondo(self) -> None:
        if not self.media_player_wav.isPlaying():
            self.media_player_wav.play()

    def parar_musica_fondo(self) -> None:
        if self.media_player_wav.isPlaying():
            self.media_player_wav.stop()
```

★ Notas sobre sonidos

- Al reproducir un MP3 puede aparecer el *warning Could not update timestamps for skipped samples*. Se debe a cómo está comprimido el formato y **no afecta en nada**.
- En **Windows**, si el *path* absoluto supera los **260 caracteres**, el sonido no se reproduce. Evita carpetas muy profundas.
- Mucho código en internet usa PyQt5 (`QMediaContent` y `setMedia`). En PyQt6 se usa `QAudioOutput` y `setSource`.

9 Errores frecuentes y hoja de referencia

9.1 Errores frecuentes: síntoma → causa → solución

Síntoma	Causa probable	Solución
La ventana se congela y no responde	Un `sleep`, un `recv` o un ciclo largo en un *slot*	`QTimer` para lo periódico, *thread* para lo que bloquea
`Segmentation fault`, o *widgets* que se portan al azar	Un *thread* modifica *widgets* directamente	Regla de oro: el *thread* emite señales
`QThread: Destroyed while thread " is still running`	El `QThread` solo estaba en una variable local, o se reasignó mientras corría	Guarda la referencia en un atributo o lista; revisa `isRunning()`
Un método de mi `QThread` congela la GUI	Solo `run` corre en el *thread* nuevo; los *slots* corren donde vive el objeto	El trabajo pesado va dentro de `run`
Se emite una señal pero el método nunca se ejecuta	El receptor se creó dentro de un `run`, en un *thread* sin *event loop*	Crea los objetos de PyQt en el *thread* principal
Al volver al menú y jugar otra vez, todo va al doble de velocidad	Los *threads/timers* de la partida anterior siguen corriendo	Detenlos al cerrar: bandera + `wait()`, o `stop()`
`AttributeError: ... has no attribute 'exec_'`	Código de PyQt5	Usa `app.exec()`
El programa se cierra con `Aborted (core dumped)` tras una excepción	No hay *hook* de excepciones	Instala `sys.excepthook = hook`
Una imagen no carga	La terminal no está en la carpeta del *script*	`cd` a la carpeta, o *path* desde `os.path.dirname(__file__)`
`OSError: [Errno 98] Address already in use`	Otro servidor usa el puerto, o sigue reservado	Cierra el otro servidor; `SO_REUSEADDR` antes del `bind`
Mensajes cortados o pegados	`recv` sin protocolo	Largo primero (4 *bytes*) y recibir por *chunks*

9.2 Hoja de referencia

Herramienta	Qué hace	Detalle clave
`QTimer(self)`	Crea un *timer*.	El *parent* lo mantiene vivo.
`timer.timeout.connect(f)`	Conecta el método periódico.	Corre en el *thread* principal.
`setInterval(ms) / start() / stop()`	Intervalo, iniciar, detener.	Milisegundos; `start()` reinicia el conteo.
`isActive() / remainingTime()`	¿Corre? / ms restantes.	Útiles para pausar.
`QTimer.singleShot(ms, f)`	Ejecuta `f` una vez.	No cancelable (usa `setSingleShot`).
`class T(QThread): run()`	Define un *thread*.	Solo `run` corre en el *thread* nuevo.
`start() / wait() / isRunning()`	Iniciar / esperar / ¿corre?	Equivalen a `start`/`join`/`is_alive`.
`finished / started`	Señales del `QThread`.	Al terminar / iniciar `run`.
`requestInterruption()`	Pide detenerse.	Revisar con `isInterruptionRequested()`.
`QMutex: lock() / unlock()`	Sección crítica.	`unlock()` en un `finally`.
`obj.moveToThread(qthread)`	Cambia la afinidad.	Patrón *worker*.
`closeEvent(evento)`	Al cerrar una ventana.	Detener *threads*; `evento.accept()`.
`app.aboutToQuit`	Señal antes de salir.	Conectar al `cerrar`/`detener` del *back-end*.
`sys.excepthook = hook`	Ver errores sin cerrar.	PyQt revisa *si* lo cambiaste.
`evento.isAutoRepeat()`	¿Tecla repetida?	`False` solo la primera vez.
`enviar / recibir (protocolo)`	Mensaje con largo.	4 *bytes* `big` + *chunks*.
`sock.shutdown(SHUT_RDWR)`	Despierta un `recv` bloqueado.	Antes de `close()` y `wait()`.
`setsockopt(..., SO_REUSEADDR, 1)`	Reutiliza el puerto.	Antes del `bind`.

✓ Checklist de una GUI con concurrencia sana

1. ¿Algún *slot* tiene un `sleep`, un `recv` o un ciclo largo? ⇒ `QTimer` o *thread*.
2. ¿Algún *thread* toca un *widget*? ⇒ cámbialo por un `emit`.
3. ¿El trabajo pesado del `QThread` está **dentro de `run`**? ¿Creé los objetos de PyQt en el *thread* principal?
4. ¿Guardé la **referencia** de cada `QThread` (atributo o lista)? ¿Reviso `isRunning()` antes de crear otro?
5. ¿Detengo *threads* y *timers* al cerrar (`closeEvent` / `aboutToQuit`) con bandera + `wait()` o `stop()`?
6. En red: ¿protocolo con largo? ¿`recv` en un *thread*? ¿manejo `ConnectionError`? ¿`shutdown` al cerrar?
7. ¿Instalé `sys.excepthook`? ¿Mi código es de **PyQt6** y no de PyQt5?



Fin del resumen • ¡Que tu event loop *nunca se detenga!*