Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Puebla

Implementación de redes de área amplia y servicios distribuidos (Gpo 201)

Reporte de actividad: GUI Procesamiento distribuido

Fernando Jiménez Pereyra	A01734609
Daniel Munive Meneses	A01734205
Alejandro López Hernández	A01733984

3 de mayo de 2023

Descripción

¿Qué se va a hacer?

La finalidad de esta actividad es desarrollar un interfaz gráfica para el cluster de procesamiento de imágenes, realizado en la actividad anterior. La imagen de entrada proporcionada por el usuario debe estar en blanco y negro, ademas que se debe proporcionar el numero de imagnes porcesada desea el usuario, siendo el maximo de 50. Y se espera que la salida sean las imágenes procesadas dentro de la carpeta compartida.

La evidencia requerida para la actividad es un reporte detallando la forma en que se desarrolló la interfaz gráfica, así como los procedimientos que lleva a cabo la interfaz para su correcto funcionamiento.

¿Qué herramientas se emplearán?

Software

Para diseñar la solución se hizo uso del lenguaje python 3 para el desarrollo de la interfaz, junto con los siguientes módulos:

- PyQt5: Nos permite el desarrollar ventanas, así como obtener entradas del usuario mediante drag & drop, como por diálogos.
- sys: Se utilizó para poder hacer uso de la interfaz gráfica.
- os: Nos permite interactuar directamente con el sistema operativo, mediante una shell. Se usó con el fin de mandar a llamar el procesamiento de las imágenes en el clúster.
- puremagic: Fue utilizado para realizar la comprobación del formato de las imágenes dadas por el usuario, mediante los headers del archivo.

Hardware

A su vez se hizo uso de la misma arquitectura física, formada de un cluster de 3 máquinas virtuales linux, con las siguientes características:

- Master:
  - 8 de Ram
  - 2 CPUs
  - 20 GB de almacenamiento

- Client 1
  - 4 de Ram
  - 2 CPUs
  - 20 GB de almacenamiento

- Client 2
  - 20 de Ram
  - 5 CPUs
  - 10 GB de almacenamiento

Metodología

Ejecución de la interfaz

Para ejecutar la interfaz gráfica se requiere primeramente de tener instalados las dependencias, lo cual podría requerir que sean instaladas con el manejador de paquetes de la distribución en vez de pip, ya que la interfaz gráfica requiere correr con privilegios, a su vez que el ejecutable que va a correr el cluster se encuentre en el mismo directorio donde nos encontramos.

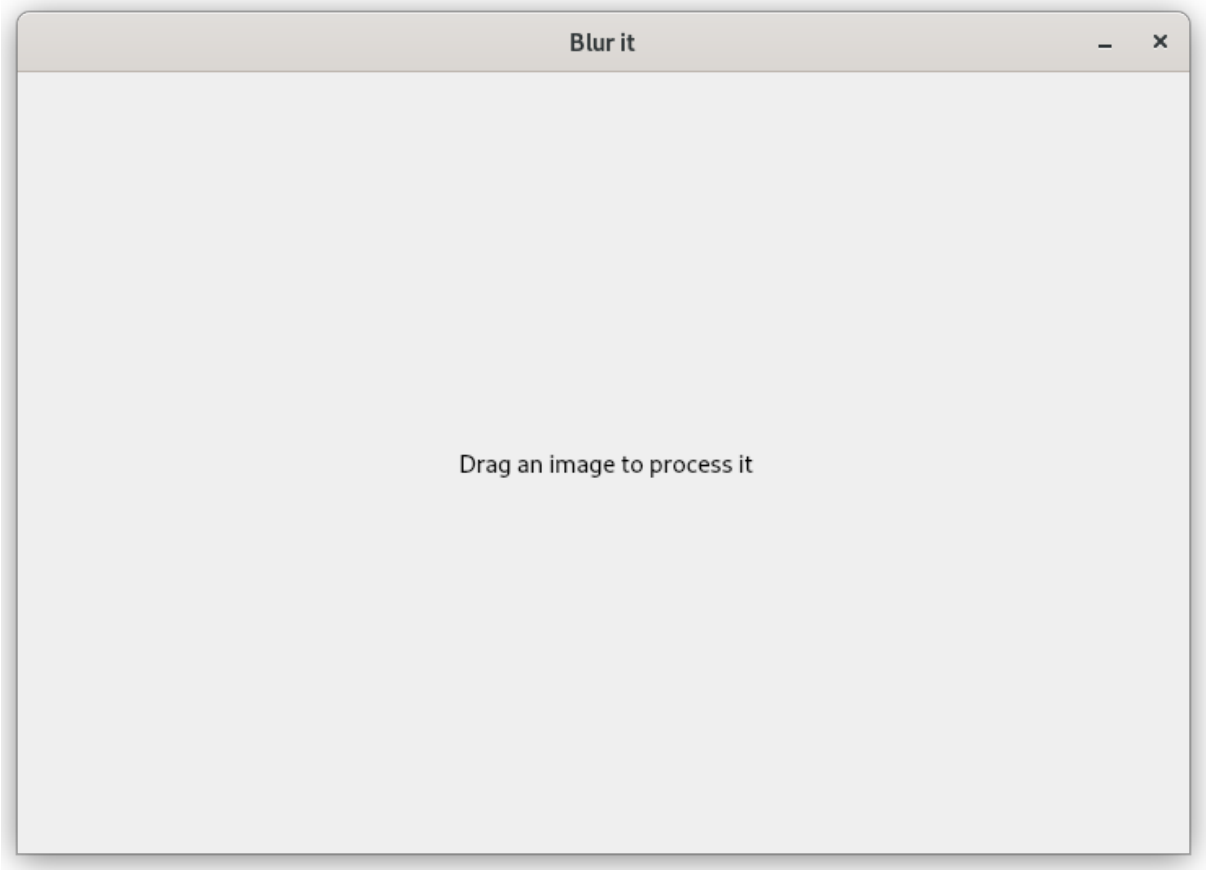
Una vez instaladas las dependencias se debe estar posicionado en el directorio /mirror/mpiu/ (este paso es válido para nuestra arquitectura, ya que nuestro ejecutable para el cluster se encuentra en este directorio)

Antes de ejecutar la interfaz debemos comprobar que nuestro usuario cuente con un display, para hacerlo únicamente tendremos que ejecutar el comando “echo $DISPLAY”, en caso de retornar cualquier display podemos seguir con ejecución de la interfaz, en caso de no retornar nada tendremos que cambiarnos a un usuario que si posea display. Esta es la razón por la que nuestra interfaz hace uso de privilegios, debido a que el usuario mpiu necesario para ejecutar el procesamiento en el cluster no posee un display, por lo que corremos la interfaz con con privilegios para cambiar internamente de usuario.

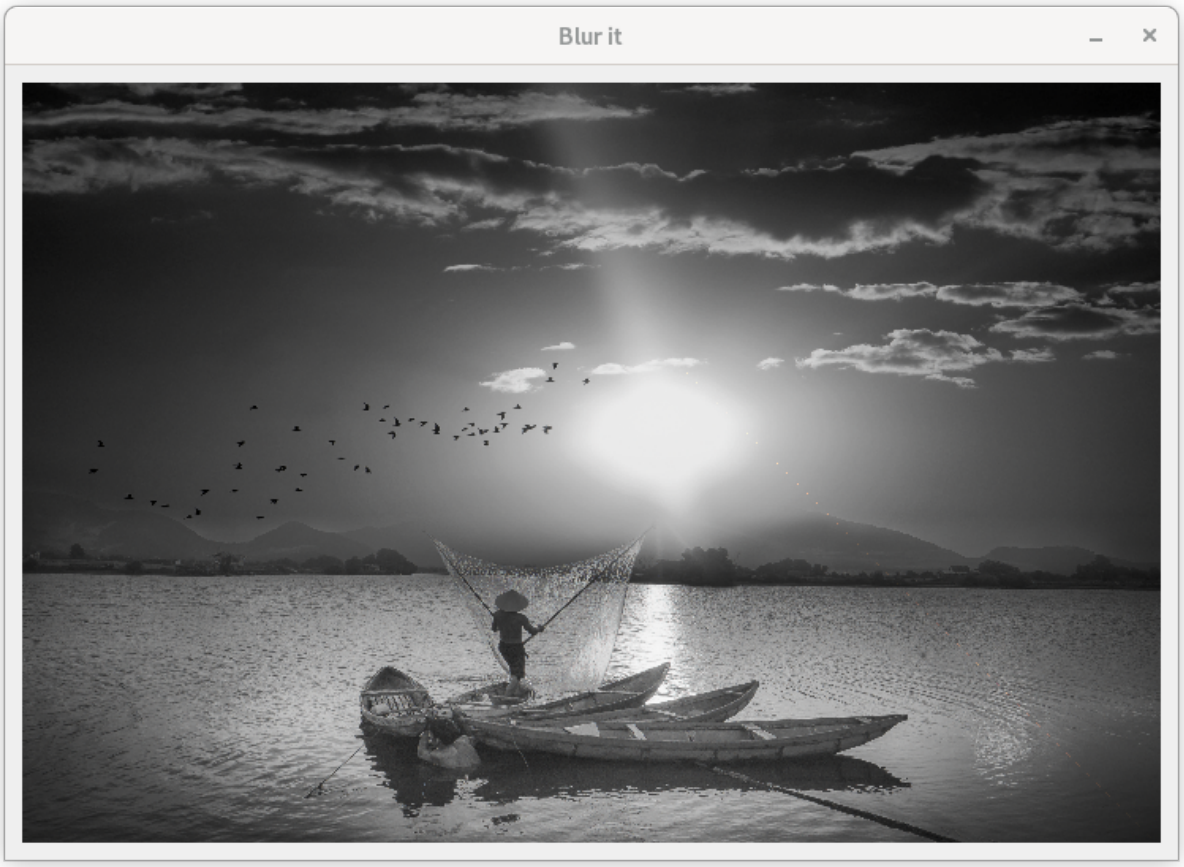
Funcionamiento de la interfaz

La interfaz gráfica fue programada en python 3 junto con Qt, mediante el módulo PyQt5. El diseño de la interfaz se orientó a ser fácil de usar, e intuitivo, sin elementos extras innecesarios, proporcionando elementos al usuario a medida que lo va necesitando para interactuar con la interfaz.

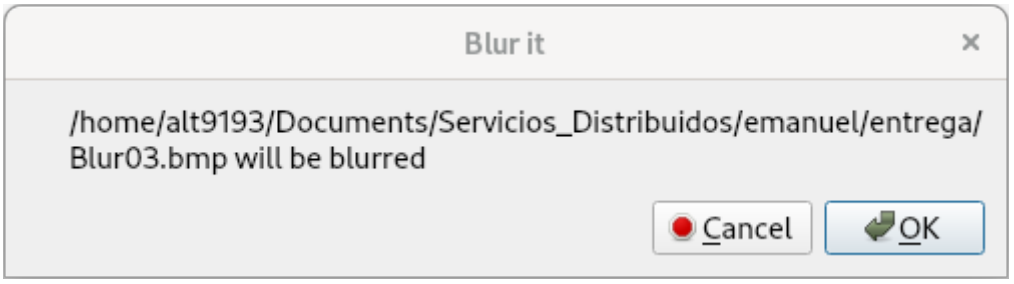
Inicialmente al correr la interfaz se muestra un lienzo en blanco, el cual nos indica que arrastremos una imagen para ser procesada.



Una vez que proporcionamos una imagen en blanco y negro en formato bmp, esta se muestra en el lienzo, y nos pedira una confirmacion para procesarla. En caso de no ser una imagen en formato bmp, la interfaz ignorara la imagen a la espera de una imagen en formato bmp.

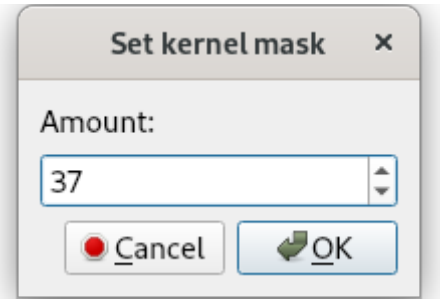


Junto con el cambio del lienzo la interfaz nos mostrará un diálogo en donde podemos escoger procesar la imagen, o no. En caso de cancelar la interfaz no seguirá con el procesamiento de la imagen, mientras que si damos a aceptar la interfaz continuará con los siguientes pasos.

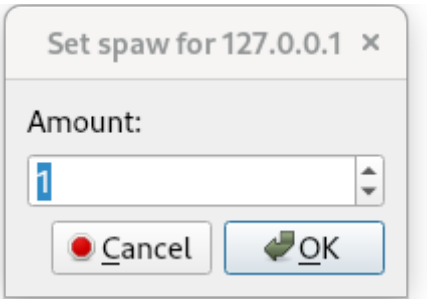


Nos aparecerá un diálogo en donde nos pide que indiquemos el número de máscaras que queremos que se aplique al procesamiento de la imagen, cabe recordar que el número de máscaras también se refleja en el número de imágenes finales, en donde cada imagen tendrá una máscara diferentes.

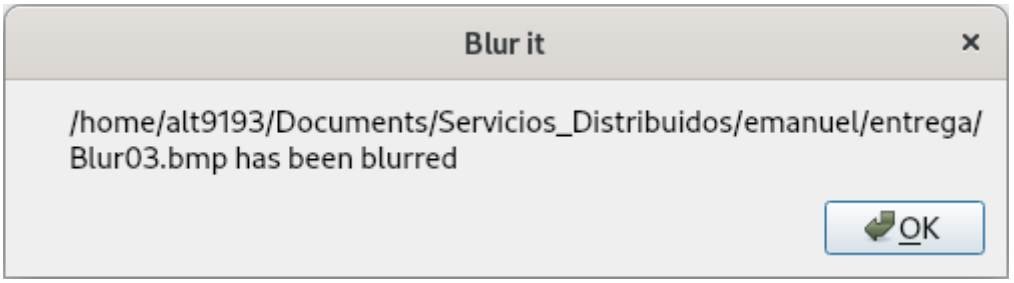
El número de máscaras puede ir desde 1 como mínimo, y hasta 50 como máximo.



Posteriormente la interfaz realiza una comprobación de conectividad entre los hosts presentes en el archivo “/mirror/mpiu/machinefile”. Una vez comprobado los hosts le pedirá al usuario que asigne la cantidad de procesos a los hosts disponibles.



Finalmente la interfaz mandará a llamar el procesamiento de la imagen en el cluster con los parámetros asignados, Y al finalizar mostrará un diálogo con un mensaje de confirmación, y mostrará la imagen resultante con la máscara más alta en el lienzo.




Blur it

Funcionamiento del código

Dependencias

```
from PyQt5.QtWidgets import *

from PyQt5.QtCore import *

from PyQt5.QtGui import *


import sys

import os

import puremagic
```

Primeramente importamos las dependencias que utilizamos en el código. Las dependencias que inician con “PyQt5” las utilizamos para crear la interfaz gráfica, mientras que “sys” es utilizada al momento de correr el programa para generar una instancia, por otra parte “os” es utilizada para comunicarnos directamente con el sistema operativo mediante una shell con el fin de mandar a llamar el procesamiento de las imágenes, y finalmente “puremagic” es utilizada para comprobar que las imágenes proporcionadas por el usuario se encuentre en formato bmp.

Visor de imagenes

```
class ImageLabel(QLabel):

    def __init__(self):

        super().__init__()

        self.setAlignment(Qt.AlignCenter)

        self.setText("\n\n Drag an image to process it \n\n")


    def setPixmap(self, image):

        scaled = image.scaled(720, 480, Qt.KeepAspectRatio)

        super().setPixmap(scaled)
```

Creamos una clase llamada “ImageLabel” a partir de “QLabel” la cual nos servirá como lienzo para mostrar las imágenes. La clase en cuestión tiene los atributos de centrar su contenido, y mostrar un texto con las instrucciones iniciales del uso de la interfaz. A su vez posee la función setPixmap, la cual nos permite ponerle una imagen al lienzo con una resolución base de 720x480, pero manteniendo la relación de aspecto, esta función hace uso de de objeto de tipo pixmap, el cual es la forma en la cual PyQt maneja las imágenes.

Clase principal

```
class MainWidget(QWidget):

    def __init__(self):

        super().__init__()

        self.setWindowTitle("Blur it")

        self.resize(720, 480)

        self.setAcceptDrops(True)



        mainLayout = QVBoxLayout()

        self.photoViewer = ImageLabel()

        mainLayout.addWidget(self.photoViewer)

        self.setLayout(mainLayout)



        self.kernel = 0

        self.hosts = []

        self.spaws = []
```

Creamos una clase llamada “MainWidget” a partir de “QWidget” con el fin de poder modificar su layout. Entre sus atributos se encuentra la posibilidad de aceptar drag & drop (self.setAcceptDrops(True)), tener un layout modificable al cual le asignamos la clase anterior para mostrar las imágenes, así como variables que nos servirá para realizar el procesamiento de imágenes como son las siguientes:

- kernel: Almacena el parámetro de número proceso a ejecutar, lo cual se va a traducir en el número de máscara a ejecutar dentro del programa escrito en C.
- hosts: Almacena los hosts disponibles
- spawn: Almacena el número de procesos asignados a cada host por el usuario.

Funciones de la clase principal

```
def dragEnterEvent(self, event):
        if event.mimeData().hasUrls():
            event.accept()
        else:
            event.ignore()
```

La función “dragEnterEvent” permite el proceso de drag & drop, ya que valida que el contenido posea una URL, para aceptar la interacción, o rechazarla en caso de no tenerla.

```
def dropEvent(self, event):
        files = [u.toLocalFile() for u in event.mimeData().urls()]
        for f in files:
            isBMP = checkIfIsBMP(f)
            if isBMP:
                self.photoViewer.setPixmap(QPixmap(f))
                if blurItMessage(f):
                    self.setImageToBlur(f)
                    # Set numero de kernel
                    self.setKernel()
                    self.testHosts()
                    if len(self.hosts) > 0:
                        for i in self.hosts:
                            self.setSpawFoHost(i)
```

```
                    else:
                        message("No connected hosts found")
                    hosts = ""
                    for i, host in enumerate(self.hosts):
                        hosts += host + ":" + str(self.spaws[i])
                        if i < len(self.hosts)-1:
                            hosts += ","
                        print("{} will be blurred in {} masks".format(f,
self.kernel))
                    self.blurImage(f, hosts)
```

La función “dropEvent” recibe la URL de la interaction de drag & drop, una vez recibida una URL, válida si se trata de una imagen en formato bmp utilizando la función “checkIfIsBMP”, en caso ser una imagen bmp la pone en el lienzo, y posteriormente consulta al usuario si desea procesar la imagen mediante la función “blurItMessage”, en caso de ser aprobado por el usuario se crea un enlace simbólico hacia la imagen mediante la función “setImageToBlur”, posteriormente consulta el número de máscaras a procesar mediante la función “setKernel”, una vez dado el número de máscaras la interfaz comprueba la conectividad con los hosts mediante la función “testHosts”, por cada host accesible le consulta al usuario cuantos procesos desea asignarle a dicho host mediante la función “setSpawForHost”, en caso de no haber ningún host disponible muestra un diálogo con el problema. Una vez asignado el número de proceso a cada host la interfaz manda a llamar el procesamiento de las imágenes con la función “blurImage”

```
def setKernel(self):
                i, okPressed = QInputDialog.getInt(self, "Set kernel
mask","Amount:", 1, 1, 50, 1)
        if okPressed:
            self.kernel = i
```

La función “setKernel” genera un diálogo en el cual el usuario puede indicar el número de máscaras a procesar, entre 1 y 50, y una vez dado se almacena en la variable kernel.

```
def testHosts(self):
        machinefile_path = "/mirror/mpiu/machinefile"
        machinefile = open(machinefile_path, "r")
        for host in machinefile:
            if host != "\n":
                host = host.split(":")[0]
                if os.system("ping -c 4 " + host) == 0:
                    self.hosts.append(host)
        machinefile.close()
```

La función “testHosts” realiza una comprobación de la conectividad de los hosts presentes en el archivo “/mirror/mpiu/machinefile” mediante 4 pings, en caso de haber al menos un ping exitoso por host se considera accesible.

```
def setSpawFoHost(self, host):
            i, okPressed = QInputDialog.getInt(self, "Set spaw for {}".format(host),"Amount:", 0, 0, 50, 1)
        if okPressed:
            self.spaws.append(i)
```

La función “setSpawFoHost” genera un diálogo para consultar el número de procesos a asignar a un host.

```
def setImageToBlur(self, file):
        # Comprueba si existe el enlace simbolico
            os.system("if [ -f /mirror/GrayScale.bmp ]; then rm /mirror/GrayScale.bmp; fi")
        # Crea un enlace simbolico
        os.system("ln -s {} /mirror/GrayScale.bmp".format(file))
```

La función “setImageToBlur” comprueba si existe un archivo “/mirror/GrayScale.bmp” y de existir lo elimina, posteriormente genera un enlace simbólico de la imagen proporcionada por el usuario hacia “/mirror/GrayScale.bmp”

```
def blurImage(self, file, hosts):
       # Corre el codigo
         status = os.system("su mpiu bash -c \"mpiexec -n {} -host {}
./mpi\"".format(self.kernel, hosts))
       if status == 0:
           message(file + " has been blurred")
           resultPath = "/mirror/mpiu/images/"
           result = resultPath + "blur_" + str(self.kernel) + ".bmp"
           self.photoViewer.setPixmap(QPixmap(result))
```

La función “blurImage” manda a llamar el procesamiento de la imagen con los parámetros dados por el usuario mediante una shell usando el módulo “os”, en caso de haber finalizado el proceso exitosamente la interfaz informa al usuario mediante un diálogo, y actualiza la imagen en el lienzo.

Otras funciones

```
def blurItMessage(file):
   alert = QMessageBox()
   alert.setWindowTitle("Blur it")
   alert.setText(file + " will be blurred")
   alert.setStandardButtons(QMessageBox.Ok | QMessageBox.Cancel)

   return_value = alert.exec()
   if return_value == QMessageBox.Ok:
       return True

   else:
       return False
```

La función “blurItMessage” genera un diálogo para preguntarle al usuario si desea procesar la imagen dada.

```
def message(message):
   alert = QMessageBox()
   alert.setWindowTitle("Blur it")
   alert.setText(message)
   alert.exec_()
```

La función “message” genera un diálogo con el mensaje dado.

```
def checkIfIsBMP(file):
    format = puremagic.from_file(file)
    if format == ".bmp":
        return True
    else:
        alert = QMessageBox()
        alert.setText(file + " is not an image in BMP format")
        alert.exec_()
        return False
```

La función checkIfIsBMP utiliza el modulo “puremagic”, para analizar los headers del archivo y comprobar de manera sencilla si se trata de una imagen en formato bmp, de serlo regresa un True, en el caso contrario genera un diálogo informando del hecho, y regresa un False.

Creación de la instancia

```
if __name__ == '__main__':
    app = QApplication(sys.argv)
    ui = MainWidget()
    ui.show()
    sys.exit(app.exec_())
```

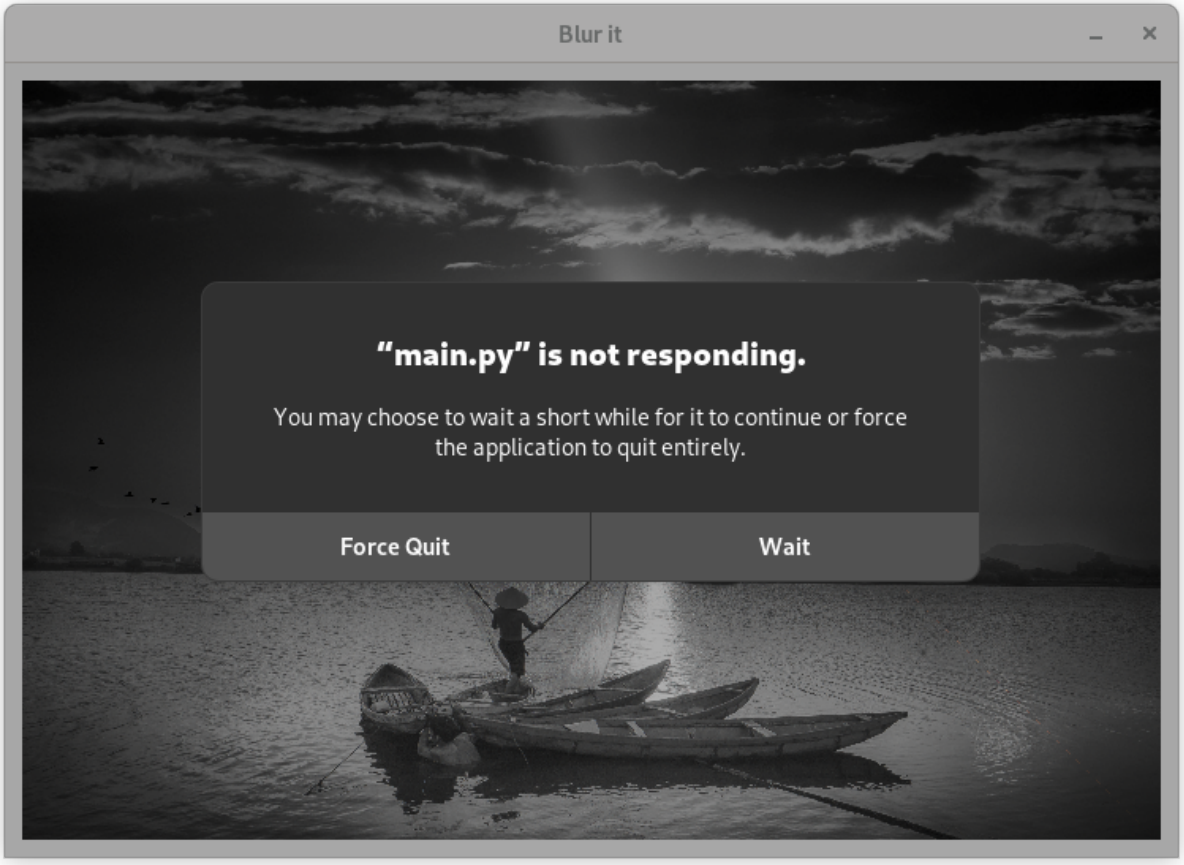
Al final del script de python se encuentran las siguientes líneas que tiene como fin generar la instancia del programa, que se describirán en orden:

1. Se trata de una convención de python para asegurarse de que solo será ejecutado en caso de tratarse de un programa principal, y no de un módulo importado.
2. Genera un bucle de eventos de PyQt.
3. Genera una instancia para la clase principal.
4. Muestra la ventana principal.
5. Ejecuta el bucle principal de eventos de PyQt, y espera a que la aplicación se cierre antes de finalizar, y utiliza la función “sys.exit()” para asegurarse de liberar los recursos y cerrar de forma adecuada.

Resultados

La interfaz es capaz de utilizar el cluster configurado, y ejecutar el procesamiento de imágenes con los parámetros pasados por el usuario, haciendo más fácil su uso y siendo algo más similar a las herramientas que usamos en nuestro dia a dia. Mientras que los resultados del procesamiento de las imágenes por parte de la interfaz es el mismo que el de la actividad pasada, ya que la interfaz únicamente se encarga de mandar a llamar el procesamiento de las imágenes con los parámetros dados por el usuario, sin embargo todavía se podría trabajar para mejorar los resultados.

Entre las mejoras que se podría implementar estaría el desarrollar más la interfaz para que se visualiza de mejor manera los resultados del procesamiento de las imágenes, y no únicamente la imagen con la máscara más grande. Además de utilizar QThread con el fin de realizar la llamada del procesamiento de las imágenes de forma paralela, y evitar que se congele la interfaz, si bien no es un problema que impida su uso, ya que uno puede esperar a que termine el procesamiento e ignorar la advertencia, no es lo más amigable para el usuario, y podría dar la impresión de que el programa fallo en plenitud.



Conclusiones

Alejandro López Hernández:
La inclusión de una interfaz gráfica fue sin duda una implementación necesaria para poder aterrizar lo de actividades pasadas a un entorno un poco más cercano a los productos finales que emplean este tipo de tecnologías. Si bien los resultados son similares, se incluyó también una validación para poder redistribuir los procesos en caso de que alguno de los nodos presente una conexión fallida.

Daniel Munive Meneses:
En conclusión, la interfaz tiene la capacidad de procesar imágenes utilizando el cluster configurado, lo que facilita su uso y se asemeja más a las herramientas cotidianas. Aunque los resultados son los mismos que en la actividad anterior, se pueden implementar mejoras en la interfaz para visualizar de manera más clara los resultados y utilizar QThread para evitar que se congele la interfaz, lo que mejoraría la experiencia del usuario.

Fernando Jimenez Pereyra:

Considero que la agregación de una interfaz gráfica hace más fácil el uso de sistema ya creado, ya que permite de una manera más sencilla el asignar una imagen a procesar, asignar el nivel de procesamiento, la comprobaciones de los nodos, así como la visualización del resultado. Ya que permite realizar todas estas acciones desde un punto

central y reduciendo la probabilidad de equivocaciones al tener el proceso automatizado y con comprobaciones.

Por otra parte considero que la interfaz es mejorable, ya que se podría mejorar la navegación, así como evitar que se congele usando hilos, a su vez se podría reescribir el código para hacerlo más limpio, y eliminar algunas faltas de ortografía, lo cual no se hizo para evitar posibles incidentes.