IMPLEMENTACIÓN E INSTALACIÓN DEL PROYECTO 1 – MIDDLEWARE

CESAR ANDRÉS GARCÍA POSADA

DANIEL GARCÍA GARCÍA

JUAN CAMILO GUERRERO ALARCÓN

UNIVERSIDAD EAFIT

INGENIERÍA DE SISTEMAS

TÓPICOS ESPECIALES DE TELEMÁTICA

MEDELLÍN, ANTIOQUIA

2021

TABLA DE CONTENIDO

INTRODUCCIÓN

El middleware tiene una gran importancia para la conexión de las aplicaciones, con otras aplicaciones, procesos o servicios, en otras palabras, es ese puente que permite establecer la comunicación y el paso de información para que los programas de software funcionen correctamente. Con el objetivo de afianzar los conocimientos del middleware a través de la práctica, se nos ha propuesto desde la materia Tópicos de Telemática de la Universidad EAFIT, implementar un Middleware orientado a mensajes (MOM), donde tenemos clientes publicadores que envían mensajes y otros consumidores que los reciben, dichos mensajes pueden ser enviados y recibidos bajo un modelo de colas o canales, siendo el primero de ellos el que reparte los mensajes entre los consumidores conectados, a manera de repartición de tareas, y el segundo el que envía el mismo mensaje a todos los consumidores conectados en el momento en que son enviados. Los detalles correspondientes a la implementación del proyecto los dejamos plasmados para cualquier persona a quien le pueda interesar, en los siguientes puntos del presente documento.

IMPLEMENTACIÓN

Para la implementación del middleware decidimos dividir el proyecto en 4 clases, primero tenemos client.py que gestiona las operaciones que puede hacer un usuario publicador, se conecta con el server como cliente mediante sockets, luego tenemos consumidor.py que gestiona las operaciones que puede hacer un usuario consumidor, conectándose también como cliente mediante el uso sockets, en tercer lugar, tenemos server2.py que es como tal el servidor encargado de tomar decisiones de acuerdo a las opciones ingresadas por los usuarios, gestiona todo el MOM haciendo uso de la última clase Aplicación.py, esta última clase se encarga de definir los métodos para las acciones que se toman en el MOM cumpliendo los requerimientos del proyecto, tiene entonces la función de gestionar toda la información de las colas (arreglos bajo el modelo de cola o canal), los estados de estas anteriores, estados de los usuarios, los mensajes y encriptar las claves de acceso a los canales o colas. Todo esto se podrá ver más a detalle en las siguientes figuras.

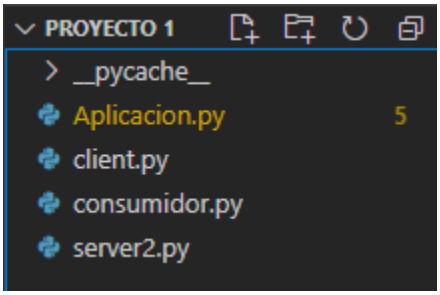


Figura 1. Lista de clases del Middleware

client.py > main

```python
import socket
import constants

socketProductor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)


def main():
    print('*' * 50)
    print("Estás conectando una nueva aplicación productora al MOM\n")
    socketProductor.connect(('18.214.102.119', constants.PORT))
    tuplaConexion = socketProductor.getsockname()
    print("Tu dirección de conexión es: ", tuplaConexion)

    opcion = menuInicial()
    while opcion != "SALIR":
        if opcion == '':
            print("Opcion invalida, intenta de nuevo\n")
            opcion = menuInicial()
        elif (opcion == "CREAR-SESION"):
            usuario = input("Ingrese el usuario ")
            contrasena = input("Ingrese la contraseña ")
            envioMOM = opcion + ' ' + usuario + ' ' + contrasena
            socketProductor.send(bytes(envioMOM, constants.ENCODING_FORMAT))
            datosRecibidos = socketProductor.recv(constants.RECV_BUFFER_SIZE)
            print(datosRecibidos.decode(constants.ENCODING_FORMAT))
            opcion = menuInicial()
        elif (opcion == "INICIAR-SESION"):
            usuario = input("Ingrese el usuario ")
            contrasena = input("Ingrese la contraseña ")
            envioMOM = opcion + ' ' + usuario + ' ' + contrasena
```

Figura 2. Inicio de la clase client.py donde se gestiona el usuario publicador

client.py > ...

```
159   def menuInicial():
160       print("CREAR-SESION: Crear la sesión")
161       print("INICIAR-SESION: Inicio de sesión")
162       print("OPCION SALIR: Desconectar aplicación")
163       opcion = input("Ingrece la opcion que quiere realizar ")
164       return opcion
165
166   def menu():
167       print("OPCION CREAR-COLA: Crear una nueva cola")
168       print("OPCION LISTAR-COLA: Listado de Colas en el MOM")
169       print("OPCION BORRAR-COLA: Eliminar una Cola del MOM")
170       print("OPCION CONECTAR-COLA: Conexión a una Cola del MOM")
171       print("OPCION DESCONECTAR-COLA: Desconexión una Cola del MOM")
172       print("OPCION MENSAJE-COLA: Envio de un mensaje")
173
174       print("OPCION CREAR-CANAL: Crear una nueva cola")
175       print("OPCION LISTAR-CANAL: Listado de Colas en el MOM")
176       print("OPCION BORRAR-CANAL: Eliminar una Cola del MOM")
177       print("OPCION CONECTAR-CANAL: Conexión a una Cola del MOM")
178       print("OPCION DESCONECTAR-CANAL: Desconexión una Cola del MOM")
179       print("OPCION MENSAJE-CANAL: Envio de un mensaje")
180
181       print("OPCION SALIR: Desconectar aplicación")
182       opcion = input("Ingrese la opcion que quiere realizar ")
183       return opcion
184
185   if __name__ == '__main__':
186       main()
187
```

Figura 3. Menú de operaciones para un usuario publicador

La clase client.py inicia como nos muestra la *Figura 2*, como primer paso tenemos la importación del módulo sockets, ya que este será quién nos va permitir establecer conexión con el servidor para el envío y la recepción de mensajes, tenemos una segunda importación de un archivo *constants*, que utilizamos únicamente para recuperar el puerto al que nos queremos conectar y no estar escribiéndolo en cada clase cliente. Se puede observar que directamente en el main establecemos la conexión con el servidor, acto seguido, como nos indica la *Figura 3*, se muestra al usuario publicador las opciones que tiene disponible para ejecutar, para poder acceder a las operaciones con canales y colas, debe registrarse e iniciar sesión como nos indica el apartado de menuInicial, también presente en la *Figura 3*. Devolviéndonos a la *Figura 2*, podemos observar que después de leer la opción ingresada por el usuario, el programa entra en un ciclo de validación donde verifica cuál opción solicitó el usuario publicador, para enviársela posteriormente al servidor, esperando una respuesta de este último.

consumidor.py > main

```python
import socket
import constants

socketConsumidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)


def main():
    print('*' * 50)
    print("Estás conectando una nueva aplicación consumidora al MOM\n")
    socketConsumidor.connect(('18.214.102.119', constants.PORT))
    tuplaConexion = socketConsumidor.getsockname()
    print("Tu dirección de conexión es: ", tuplaConexion)
    opcion = menu()

    while opcion != "SALIR":
        if opcion == '':
            print("Opcion invalida, intenta de nuevo\n")
            opcion = menu()
        elif (opcion == "LISTAR-COLA"):
            envioMOM = opcion
            socketConsumidor.send(bytes(envioMOM, constants.ENCODING_FORMAT))
            datosRecibidos = socketConsumidor.recv(constants.RECV_BUFFER_SIZE)
            print(datosRecibidos.decode(constants.ENCODING_FORMAT))
            datosRecibidos = socketConsumidor.recv(constants.RECV_BUFFER_SIZE)
            print(datosRecibidos.decode(constants.ENCODING_FORMAT))
            opcion = menu()
        elif (opcion == "LISTAR-CANAL"):
            envioMOM = opcion
            socketConsumidor.send(bytes(envioMOM, constants.ENCODING_FORMAT))
            datosRecibidos = socketConsumidor.recv(constants.RECV_BUFFER_SIZE)
```

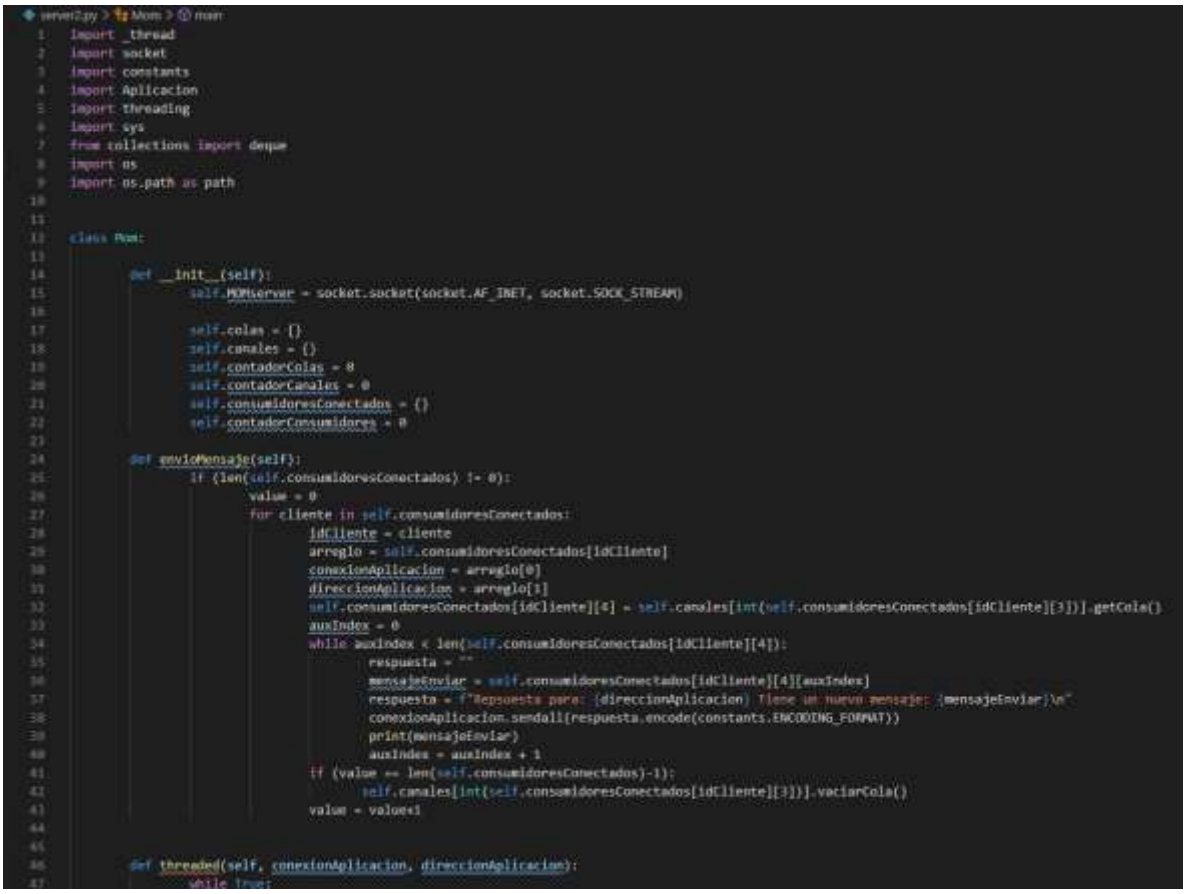
Figura 4. Inicio de la clase consumidor.py donde se gestiona el usuario consumidor

consumidor.py > ...

```
56                  opcion = menu()
57
58      socketConsumidor.send(bytes(opcion, constants.ENCODING_FORMAT))
59      datosRecibidos = socketConsumidor.recv(constants.RECV_BUFFER_SIZE)
60      print(datosRecibidos.decode(constants.ENCODING_FORMAT))
61      socketConsumidor.close()
62
63
64  def menu():
65      print("OPCION LISTAR-CANAL: Listado de Canales en el MOM")
66      print("OPCION LISTAR-COLA: Listado de Colas en el MOM")
67      print("OPCION PULL-COLA: Conexión a una Cola del MOM")
68      print("OPCION CONSUMIDOR-CANAL: Conexión a un Canal del MOM")
69      print("OPCION SALIR: Desconectar aplicación")
70      opcion = input("Ingrece la opcion que quiere realizar ")
71      return opcion
72
73  """def menu2():
74      print("OPCION PULL: Recibir tarea")
75      print("OPCION SALIR: Salir")
76      opcion = input("Ingrece la opcion que quiere realizar ")
77      return opcion"""
78
79
80  if __name__ == '__main__':
81      main()
82
```

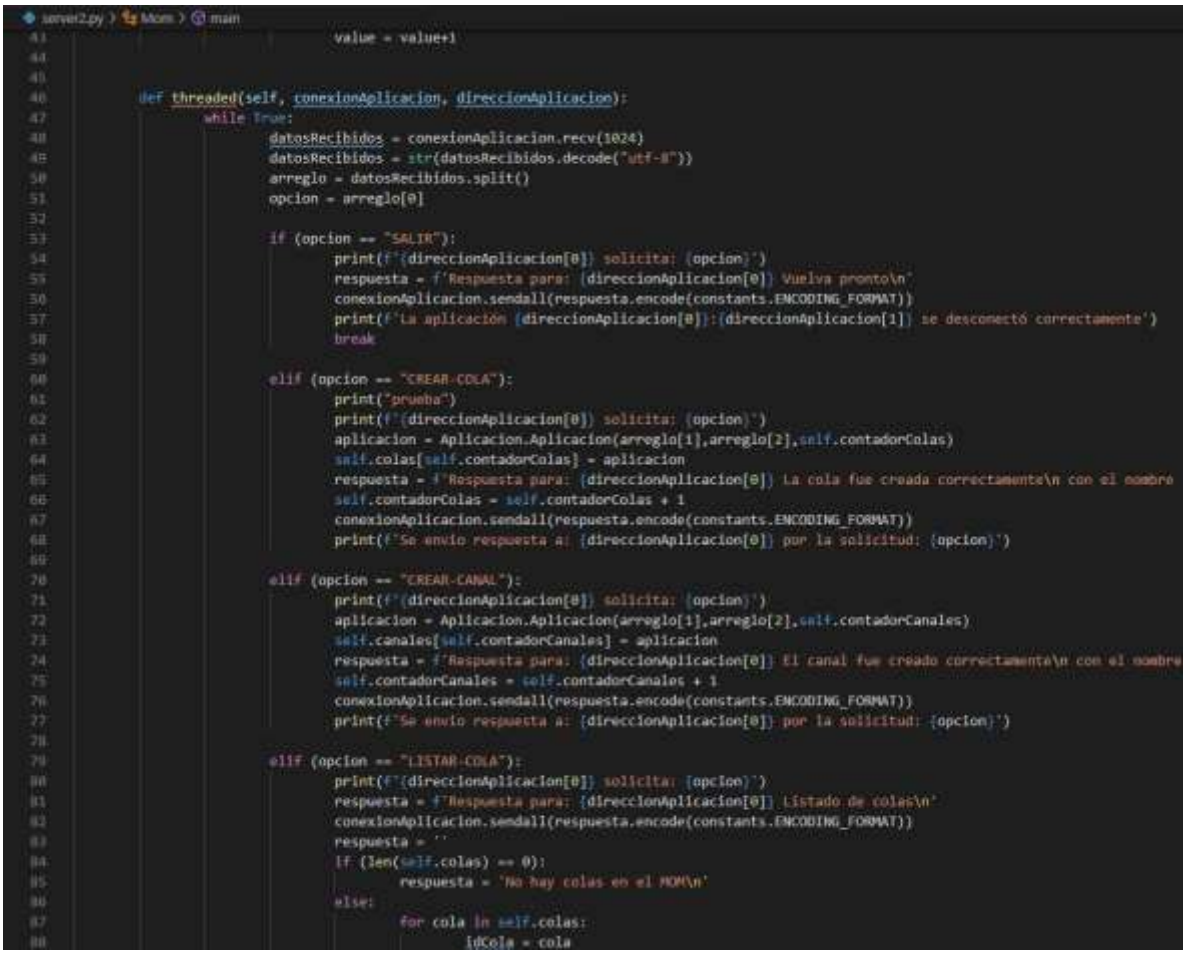
Figura 5. Menú de operaciones para un usuario consumidor

Al igual que en la clase client.py, esta clase que gestiona al usuario consumidor, inicia importando el módulo sockets observado en la *Figura 4*, para establecer conexión con el servidor e importando el archivo constants como se explicó anteriormente. La conexión con el servidor también se ejecuta directamente en el main y la gran diferencia radica en las opciones que tiene disponible un usuario consumidor. Como podemos observar en la *Figura 5*, un cliente consumidor sólo tiene la opción de listar un canal/cola o conectarse a ellos para recibir mensajes. También se establece un ciclo de validación de la opción escogida por el usuario consumidor, para ser enviada como petición al servidor, esperando una respuesta de este.



```python
import _thread
import socket
import constants
import Aplicacion
import threading
import sys
from collections import deque
import os
import os.path as path


class Mom:

        def __init__(self):
                self.MOMserver = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

                self.colas = {}
                self.canales = {}
                self.contadorColas = 0
                self.contadorCanales = 0
                self.consumidoresConectados = {}
                self.contadorConsumidores = 0

        def envioMensaje(self):
                if (len(self.consumidoresConectados) != 0):
                        value = 0
                        for cliente in self.consumidoresConectados:
                                idCliente = cliente
                                arreglo = self.consumidoresConectados[idCliente]
                                conexionAplicacion = arreglo[0]
                                direccionAplicacion = arreglo[1]
                                self.consumidoresConectados[idCliente][4] = self.canales[int(self.consumidoresConectados[idCliente][3])].getCola()
                                auxIndex = 0
                                while auxIndex < len(self.consumidoresConectados[idCliente][4]):
                                        respuesta = ""
                                        mensajeEnviar = self.consumidoresConectados[idCliente][4][auxIndex]
                                        respuesta = f"Repsuesta para: {direccionAplicacion} Tiene un nuevo mensaje: {mensajeEnviar}\n"
                                        conexionAplicacion.sendall(respuesta.encode(constants.ENCODING_FORMAT))
                                        print(mensajeEnviar)
                                        auxIndex = auxIndex + 1
                                if (value == len(self.consumidoresConectados)-1):
                                        self.canales[int(self.consumidoresConectados[idCliente][3])].vaciarCola()
                                value = value+1


        def threaded(self, conexionAplicacion, direccionAplicacion):
                while True:
```

Figura 6. Inicio de la clase server2.py donde se gestionan las peticiones del cliente



```
server2.py > Mom > main
                        value = value+1


        def threaded(self, conexionAplicacion, direccionAplicacion):
                while True:
                        datosRecibidos = conexionAplicacion.recv(1024)
                        datosRecibidos = str(datosRecibidos.decode("utf-8"))
                        arreglo = datosRecibidos.split()
                        opcion = arreglo[0]

                        if (opcion == "SALIR"):
                                print(f'{direccionAplicacion[0]} solicita: {opcion}')
                                respuesta = f'Respuesta para: {direccionAplicacion[0]} Vuelva pronto\n'
                                conexionAplicacion.sendall(respuesta.encode(constants.ENCODING_FORMAT))
                                print(f'La aplicación {direccionAplicacion[0]}:{direccionAplicacion[1]} se desconectó correctamente')
                                break

                        elif (opcion == "CREAR-COLA"):
                                print("prueba")
                                print(f'{direccionAplicacion[0]} solicita: {opcion}')
                                aplicacion = Aplicacion.Aplicacion(arreglo[1],arreglo[2],self.contadorColas)
                                self.colas[self.contadorColas] = aplicacion
                                respuesta = f'Respuesta para: {direccionAplicacion[0]} La cola fue creada correctamente\n con el nombre
                                self.contadorColas = self.contadorColas + 1
                                conexionAplicacion.sendall(respuesta.encode(constants.ENCODING_FORMAT))
                                print(f'Se envio respuesta a: {direccionAplicacion[0]} por la solicitud: {opcion}')

                        elif (opcion == "CREAR-CANAL"):
                                print(f'{direccionAplicacion[0]} solicita: {opcion}')
                                aplicacion = Aplicacion.Aplicacion(arreglo[1],arreglo[2],self.contadorCanales)
                                self.canales[self.contadorCanales] = aplicacion
                                respuesta = f'Respuesta para: {direccionAplicacion[0]} El canal fue creado correctamente\n con el nombre
                                self.contadorCanales = self.contadorCanales + 1
                                conexionAplicacion.sendall(respuesta.encode(constants.ENCODING_FORMAT))
                                print(f'Se envio respuesta a: {direccionAplicacion[0]} por la solicitud: {opcion}')

                        elif (opcion == "LISTAR-COLA"):
                                print(f'{direccionAplicacion[0]} solicita: {opcion}')
                                respuesta = f'Respuesta para: {direccionAplicacion[0]} Listado de colas\n'
                                conexionAplicacion.sendall(respuesta.encode(constants.ENCODING_FORMAT))
                                respuesta = ''
                                if (len(self.colas) == 0):
                                        respuesta = 'No hay colas en el MOM\n'
                                else:
                                        for cola in self.colas:
                                                idCola = cola
```

Figura 7. Inicio del hilo para validación de las peticiones hechas por los clientes

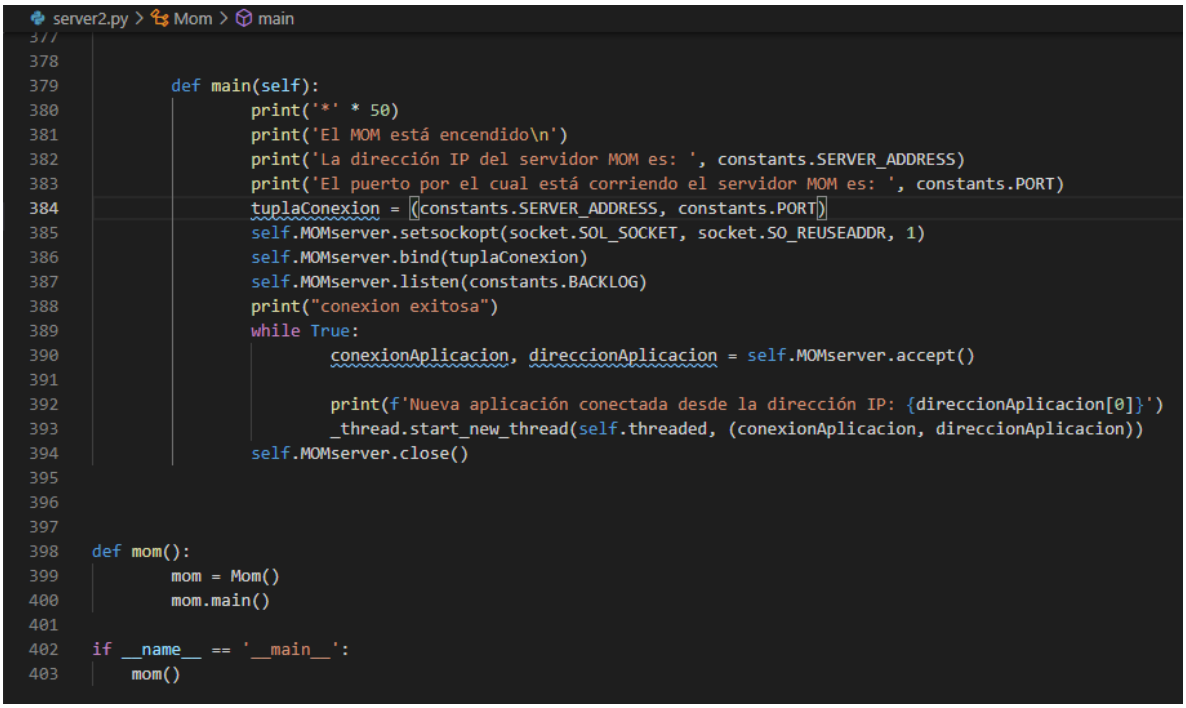


Figura 8. Main de la clase server2.py

La clase server2.py como podemos observar, es la más extensas de todas, esto porque debe gestionar todas las peticiones, tanto del usuario publicador como del usuario consumidor, y tomar decisiones al respecto. Podemos empezar analizando la *Figura 6*, donde se observa el inicio de la clase, tenemos la importación de bastantes módulos que nos van a ayudar con diferentes acciones, como el lanzamiento de hilos para soportar múltiples usuarios a la vez (*_thread*), el módulo *deque* que nos permite tener una cola con acceso desde la parte izquierda y la parte derecha, esto nos ayudará a acceder más fácil a nuestros arreglos y operar con ellos, arreglos como los mensajes o los clientes conectados, por ejemplo. Tenemos la importación de la clase *Aplicación.py* que observaremos más adelante, y por supuesto de *sockets* para establecer comunicación con los clientes.

Luego de lo anterior, podemos observar en la *Figura 7* el método que se lanza como hilo para cada usuario (*threaded*), este método es el encargado de validar cada una de las peticiones hechas por un usuario cliente, ya sea publicador o consumidor, encuentra si la petición existe y ejecuta las acciones necesarias para dar cumplimiento a la petición realizada por el cliente, apoyado gran parte de las veces de la clase *Aplicación.py* y enviándole posteriormente al cliente una respuesta a través de sockets.

Por último, observamos en la *Figura 8* el método main de la clase *server2.py*, podemos observar que inicialmente se enciende el servidor haciendo uso de sockets, y allí mismo se queda escuchando para esperar la conexión de un cliente, de manera que pueda resolver posteriormente las peticiones de este, al lanzar un hilo del método *threaded*.

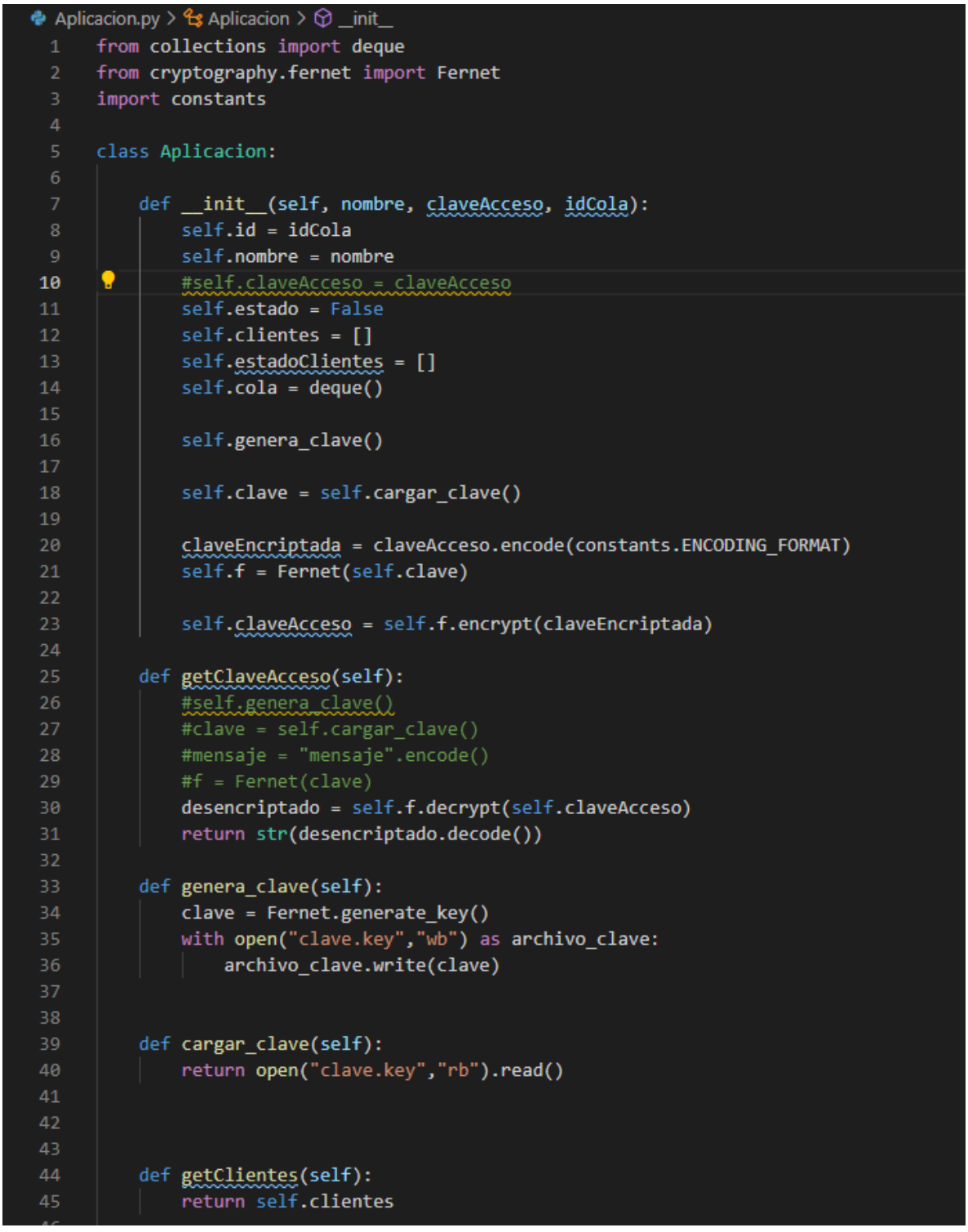


Figura 9. Inicio de la clase Aplicación.py donde se alojan gran parte de los métodos para gestionar el MOM

```
59          self.estadoClientes[i] = True
60
61      def getEstadosClientes(self):
62          return self.estadoClientes
63
64      def setEstados(self, arreglo):
65          self.estadoClientes = arreglo
66
67
68      def cambiarIndiceEnvio(self):
69          mensaje = self.cola.popleft()
70          return mensaje
71
72      #def getClaveAcceso(self):
73      #    return self.claveAcceso
74
75      def getId(self):
76          return self.id
77
78      def getNombre(self):
79          return self.nombre
80
81      def getCola(self):
82          return self.cola
83
84      def enviarMensaje(self, mensaje):
85          if (len(self.cola) < 2):
86              self.cola.append(mensaje)
87              return 1
88          else:
89              return 0
90
91      def conectar(self):
92          self.estado = True
93
94      def desconectar(self):
95          self.estado = False
96
97      def getEstado(self):
98          return self.estado
99
100     def getTamañoCola(self):
101         return len(self.cola)
102
103     def vaciarCola(self):
104         self.cola = deque()
```

Figura 10. Otros métodos de la clase Aplicación.py

Cómo última clase para la gestión del MOM tenemos a una de las más importantes: Aplicación.py. Es la encargada de realizar en sus métodos, gran parte de las funciones que le corresponden al MOM, por lo que server2.py accede a los métodos necesarios de esta clase y los resultados los devuelve al cliente en una respuesta.

Podemos empezar a analizar nuestra implementación con la Figura 9, donde podemos observar la importación de los módulos *deque* (explicado en la sección de *server2.py*) y de *Fernet*, un módulo que será el encargado de encriptar las claves de acceso ingresadas por el usuario publicador para el manejo de colas y/o mensajes. Observamos que se inicializa una cola con el nombre, id y clave de acceso, esta cola se refiere a una estructura de datos que es usada para gestionar el MOM tanto con el modo de colas como con el modo de canales, también se observan los métodos *genera_clave* y *cargar_clave* donde se encripta la clave de acceso ingresada por un usuario publicador. Posteriormente podemos observar en la Figura 10, otros métodos implementados de gran importancia para la gestión de usuarios consumidores, mensajes, colas y canales.

Nota: Las imágenes adjuntadas por cada clase, contienen las secciones de código más relevantes para la implementación del MOM, si se desea observar a detalle el código completo de cada clase, se podrá acceder a él mediante el link del repositorio adjuntado al final del presente documento.

Intervalo de p...	Protocolo	Origen	Grupos de seguridad
80	TCP	0.0.0.0/0	launch-wizard-2
80	TCP	::/0	launch-wizard-2
8080	TCP	0.0.0.0/0	launch-wizard-2
8080	TCP	::/0	launch-wizard-2
22	TCP	0.0.0.0/0	launch-wizard-2

Figura 11. Reglas y grupo de seguridad para la instancia de AWS

El anterior grupo de seguridad(launch-wizard-2), observado en la *Figura 11*, se creó utilizando el puerto 80, 8080 y el protocolo TCP, esto para poder realizar la comunicación con el servidor, este servidor es la única máquina que tenemos implementada en AWS, de manera que permitimos que cualquier cliente se conecte desde su propia máquina local como publicador o consumidor al servidor (MOM) mediante internet.

Si el cliente quiere ser productor entonces sólo es necesario que ejecute client.py y si el cliente quiere ser consumidor, sólo es necesario que ejecute consumidor.py en su máquina local, vale aclarar nuevamente, que esto también es posible gracias a los sockets que tanto se mostraron en la implementación de las clases.

INSTALACIÓN Y EJECUCIÓN

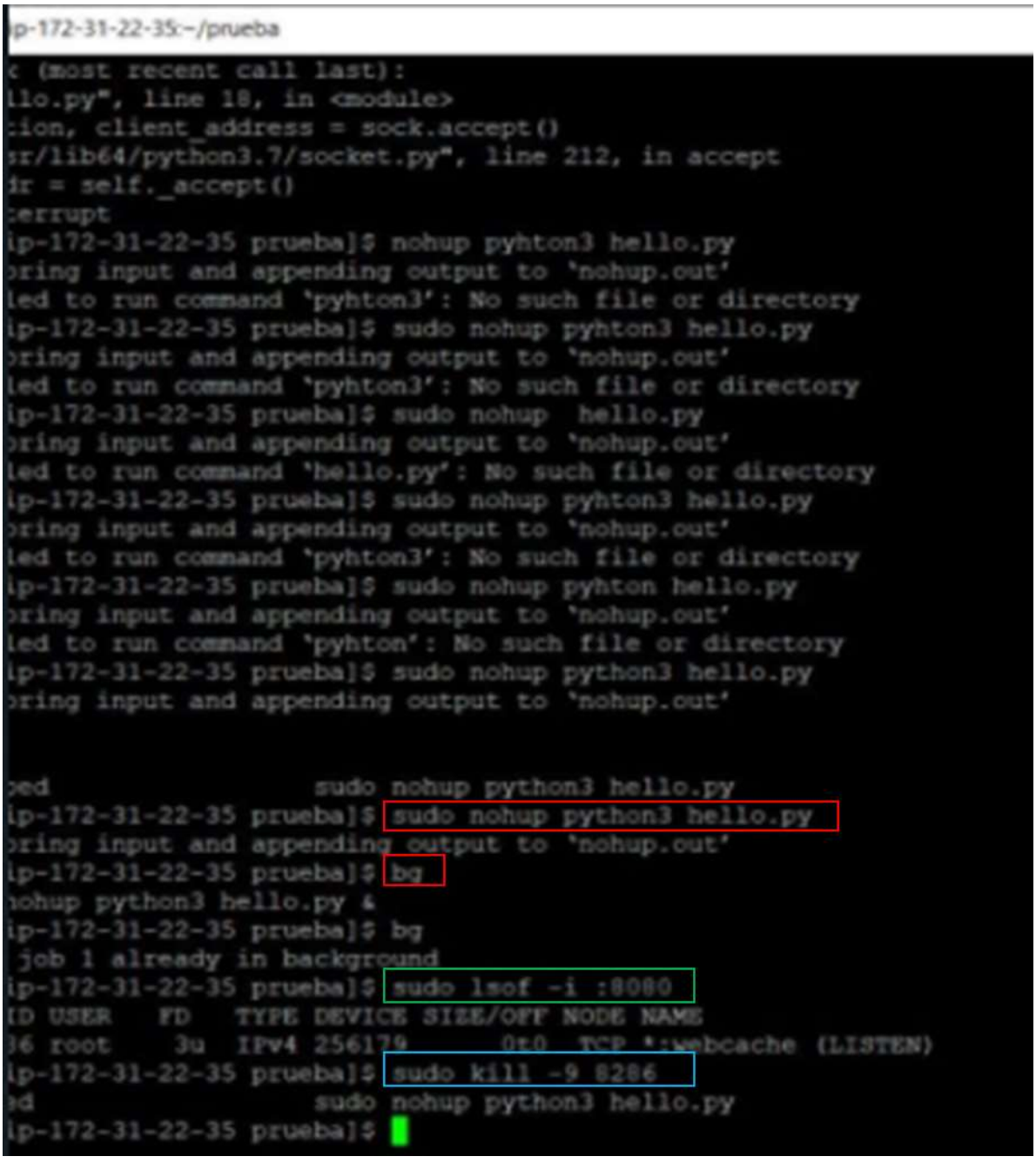


Figure 12. Comandos en la máquina virtual

En la Figura 12 encontramos señalados algunos comandos importantes para la ejecución del middleware en la máquina virtual de Linux. Los primeros dos comandos encerrados con el rectángulo rojo, nos sirven para poner a correr el servidor y que se quede siempre

corriendo, sin importar que cerremos la consola. El verde nos va servir en caso de que queramos detener el proceso que está en el puerto 8080 (puerto del servidor), ya que para esto es necesario conocer el ID de dicho proceso, y, por último, con el comando encerrado con el rectángulo azul, detenemos el proceso con el ID obtenido en el comando encerrado con el rectángulo verde.

Lo anterior en cuanto al servidor, finalmente, para los clientes la instalación y ejecución es sumamente sencilla, sólo es necesario ejecutar en la máquina local *client.py* si quiere ser un usuario productor y *consumidor.py*, si quiere ser un usuario consumidor.

ENTREGABLES ADICIONALES

Link del repositorio: https://github.com/cagarciap/practicaTelematica.git

Dirección IP: 18.214.102.119

Key de putty: El archivo .ppk se adjunta en el repositorio.

REFERENCIAS

https://rico-schmidt.name/pymotw-3/socket/tcp.html

https://www.geeksforgeeks.org/fernet-symmetric-encryption-using-cryptography-module-in-python/