

# **Documentación del Sistema de Correo**

## **1. Informe general del sistema**

### **Una vista general sobre el sistema**

El sistema presentado en este proyecto constituye una simulación que logra ser funcional y modular de cómo funciona un cliente de correos, se diseñó con la idea de utilizar programación orientada a objetos (OOP), estructura de datos y algoritmos de búsqueda.

Dentro de algunas funciones que el usuario puede realizar se encuentran:

- Registro y autenticación.
- Envío y recepción de mensajes.
- Organización de correos en carpetas y subcarpetas..
- Aplicación de reglas automáticas de clasificación.
- Utilización de una cola de prioridad para atender mensajes urgentes antes.
- Simular la interacción de una red de servidores.

A continuación en las siguientes páginas se analizaron los distintos módulos, algoritmos y funciones que se utilizaron en el proyecto.

## Módulos y responsabilidades

El sistema está formado por múltiples módulos que colaboran e interactúan entre sí de manera organizada. Cada componente tiene una responsabilidad específica y se comunica con los demás. Se le dedicará el siguiente apartado para presentar cada uno de estos e informar sus responsabilidades así como explicar cómo interactúan con los demás apartados.

- **Usuarios**

Esta es la parte que se podría considerar como el centro o núcleo del sistema, cada usuario está representado por una serie de datos personales, como las credenciales de **autenticación** (nombre, mail y usuario), cada uno de ellos posee también una estructura interna que modela su **bandeja de entrada**. Estos también cuentan con un **árbol de carpetas** que actúa como su estructura principal de almacenamiento, al cual también se le aplica una **cola de prioridad** que organiza los mensajes por escala de urgencia y una serie de **reglas** automatizadas que se aplican cuando un mensaje es recibido.

- **Mensajes**

Los **mensajes** por su parte constituyen las unidades de información que circulan entre los usuarios del sistema. Cada mensaje contiene una serie de atributos como su **remitente, destinatario, asunto y cuerpo** principalmente, luego también se les adicionan las **etiquetas, prioridad, estado de lectura** y un registro de la ruta que recorrió dentro de la red de servidores. Los mensajes son objetos autónomos, pero dependen de los usuarios y de los servidores para ser creados, enviados, procesados y almacenados. Cuando un mensaje llega al usuario destinatario, este se encarga de aplicar reglas, ubicarlo en la carpeta correspondiente y agregarlo a la cola de prioridad.

- **Carpetas**

La estructura de **carpetas** se encuentra implementada como un árbol, es fundamental para organizar los mensajes dentro de cada usuario. La **raíz** de este árbol es la que se conoce como bandeja de entrada, y a partir de ella se da la posibilidad de crear subcarpetas de manera indefinida. Este tipo de estructuramiento permite realizar **búsquedas recursivas**, incrementando la eficiencia del movimiento de mensajes. Cuando una **regla** establece que un mensaje debe moverse a una carpeta específica, el sistema navega este árbol para encontrarla o crearla de ser necesario.

- **Servidores de correo**

Los **servidores de correo** son los responsables de almacenar **usuarios** y actuar como **nodos dentro de una red de comunicación**. Cada servidor contiene un **diccionario** de usuarios registrados y recibe mensajes cuando la red determina que el destinatario pertenece a él. Su mayor función (y una de las más importantes en este proyecto) es la conexión entre usuarios remotos y la entrega correcta de los mensajes en la bandeja del destinatario.

- **Funciones de usuario**

La interacción entre el end user y el sistema se gestiona mediante la clase **funciones\_usuario**, que actúa como capa de lógica funcional. Coordinando acciones como **inicio de sesión, creación de usuarios, envío de mensajes, mover mensajes, creación de subcarpetas**, etc.. NO manipula directamente las estructuras internas, cumple como un mediador que se comunica con los objetos correspondientes, asegurando la separación de responsabilidades se realiza de manera apropiada.

- **Menú**

Por último, se encuentra el Menú que es el componente que coordina la experiencia para el end user. Este módulo se encarga de mantener la sesión activa, mostrar opciones disponibles al usuario y **dirigir cada acción hacia el componente correspondiente**. El menú, es el puente entre el end user y todas las demás capas de lógica y estructuras.

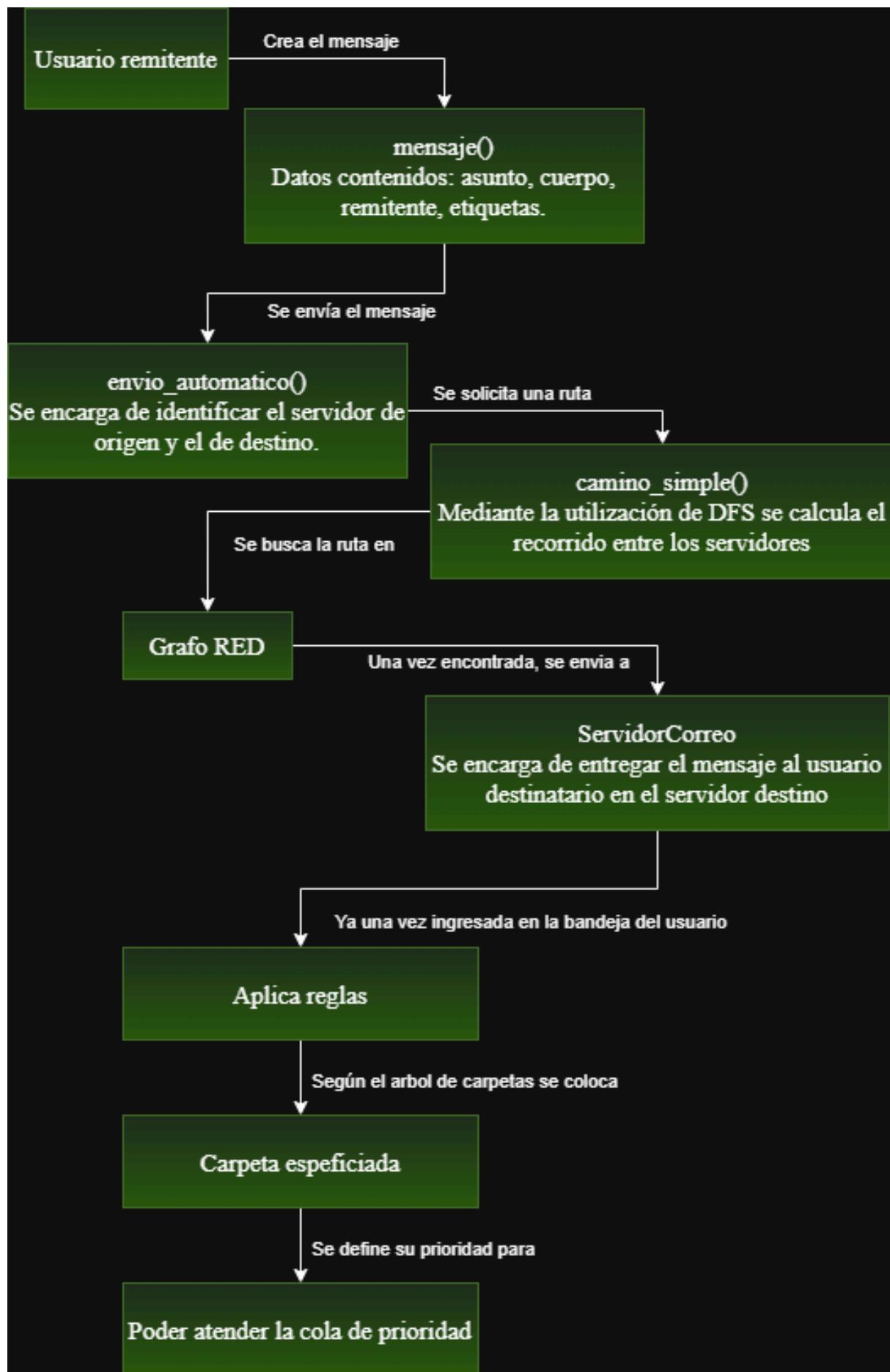
Se eligió este diseño modular con la idea de que el sistema sea, ampliable y fácil de comprender en caso de tener que realizar futuras modificaciones o correcciones.

## Flujo de datos

Para comprender el flujo de datos del sistema se debe comprender como un mensaje parte desde su creación hasta su almacenamiento final dentro del usuario destinatario.

- 1) El remitente redacta un mensaje; el sistema identifica automáticamente en qué servidor se encuentra registrado este usuario remitente y en cuál se encuentra el usuario destinatario. Una vez posee esta información, consulta la red de servidores representada por un grafo y **utiliza un algoritmo DFS para calcular un camino válido** entre ambos nodos. Esta ruta queda registrada dentro del propio mensaje.
- 2) Una vez enviado, la red dirige el mensaje hacia el servidor destino, que localiza al usuario correspondiente y activa su proceso de recepción.
- 3) Comienza el filtrado interno: el usuario aplica sus reglas automáticas designadas, que pueden modificar el mensaje mediante el agregado de etiquetas, cambiando su prioridad y moviéndolo directamente a una subcarpeta específica dentro de su árbol de carpetas.
- 4) El mensaje se almacena en la carpeta correspondiente, ya sea la bandeja principal o alguna subcarpeta generada por las reglas. En ese momento también se incorpora a la cola de prioridad del usuario, donde queda ordenado según su urgencia y su fecha.
- 5) El usuario destinatario recibe el mensaje y puede visualizarlo, buscarlo, moverlo o atenderlo desde la cola de prioridad según su decisión.

En la página siguiente se representa como queda flujo de datos desde su origen hasta el alcanzar su destinación final.



# Árbol de Carpetas

El sistema organiza los mensajes mediante un **árbol jerárquico de carpeta**, es una estructura de organización de archivos cuya forma es parecida a la de un árbol, donde en este caso las carpetas se organizan en niveles con relaciones de padre e hijo, cuyo nodo raíz distinguible en este caso es la **Bandeja de entrada**. Esta estructura replica el funcionamiento de los sistemas de correo reales, permite ordenar, buscar y clasificar mensajes de una forma más flexible.

Cada carpeta es un objeto de la clase **carpeta**, que contiene:

1. Un nombre
2. Una lista de mensajes asociados a ese nivel
3. Una lista de subcarpetas, que a su vez también son objetos **carpeta**
4. **Métodos para agregar, buscar, mover y organizar información**

De esta forma se nos permite crear estructuras anidadas y sin límites de profundidad tal como:



## Funcionamiento Interno del Árbol

Cuando el usuario decide crear una subcarpeta, o cuando una regla automática exige mover algún mensaje, el sistema va usar: **agregar\_subcarpeta(nombre)**, **obtener\_subcarpeta(nombre)**, **obtener\_o\_crear\_subcarpeta()**. Esto es así para garantizar que si la estructura debe crecer, solo lo haga donde sea verdaderamente necesario.

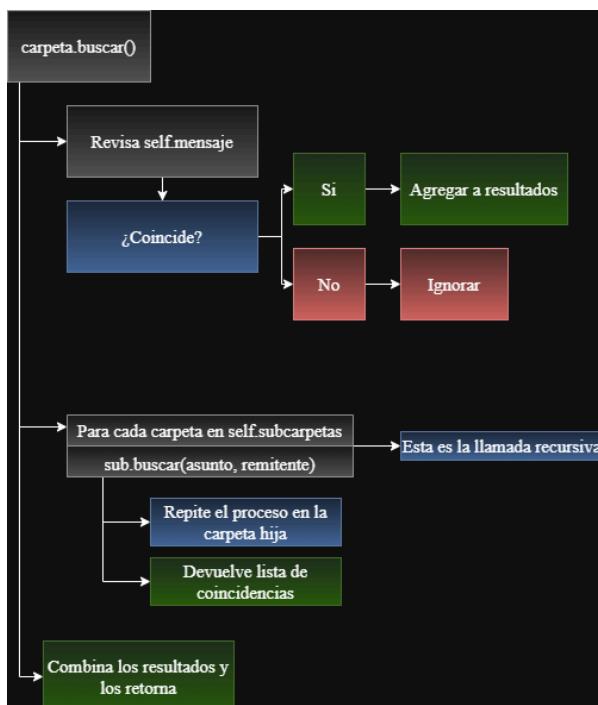
El método **agregar\_mensaje(m)** agrega un mensaje a la carpeta actual.

Dicho sea el caso que una regla automática indique que se tenga que mover algún mensaje va a utilizar: **mover\_mensaje(msg, destino)** para reorganizar automáticamente la bandeja

## Búsqueda Recursiva en el Árbol

La función **def buscar(self, asunto=None, remitente=None)** (encontrada en el módulo de carpeta, linea 49) recorre todos los mensajes de la carpeta actual, *todas* las subcarpetas y sus subcarpetas recursivamente, sin límite de profundidad. Permitiendo que funcione como un sistema de clasificación inteligente.

Ejemplo de como busca la función de nuestro código:



## Ordenamiento por Prioridad dentro del Árbol

La función **def orden\_prioridad(self)** (módulo carpeta, linea 70) lee las palabras claves del asunto, en base a dichas palabras les asigna una prioridad (En este caso, 1 a 3), reordena la carpeta en función de la urgencia y mantiene así los mensajes más importantes en la cima de la lista.

### Cola de Prioridad (heap binario)

La cola de prioridad la hemos implementado dentro de la clase usuarios con mediante: **self.cola = []**, Y funciona a través del módulo heapq, que convierte la cola en un min-heap, permitiendo obtener siempre el mensaje más urgente.

Los mensajes entran por medio del método:

```
def encolar(self, msg):
    import heapq
    k1, k2 = self.clave_prioridad(msg)
    heapq.heappush(self.cola, (k1, k2, id(msg), msg))
```

y genera una clave de prioridad, definida como:

**return (-prio, -ts)**

En la tabla a continuación se expone el resultado que esto genera.

Orden	Prioridad	Timestamp	Resultado
1°	Prioridad más alta	Mensaje más reciente	Más urgente
2°	Misma prioridad	Menos reciente	Siguiente en cola
3°	Prioridad baja	Cualquier fecha	Al final

# Aplicación de reglas

Las reglas están implementadas como una lista de diccionarios dentro de cada usuario y se agregan mediante: **agregar\_regla(condicion, accion, stop)**

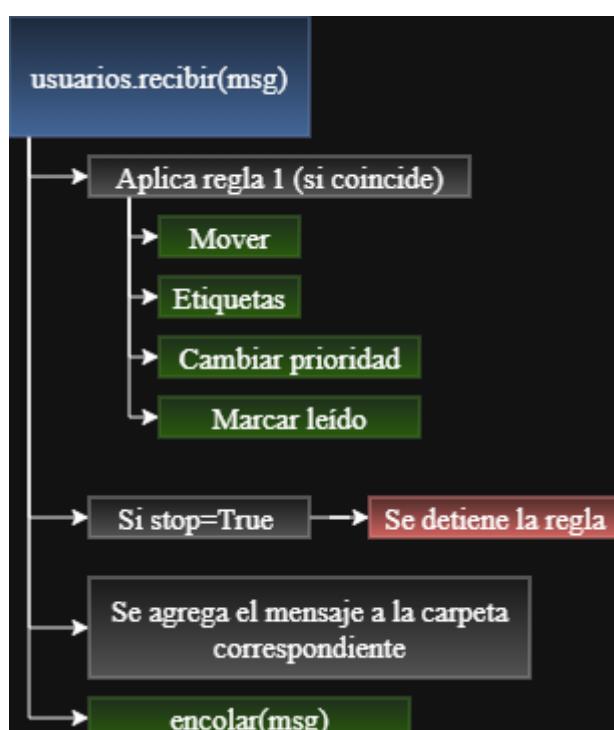
Cada regla consta de dos partes:

- Condición (¿cuándo se activa?)
- Acción (¿qué hace cuando se activa?)

**Condiciones:** la función `_matchea()` decide si una regla se ejecuta o no, mediante la evaluación o corroboración si el asunto contiene “X” (Variable deseada), remitente contiene “Y” (Otra variable deseada), o tiene “Z” etiqueta. Si no coincide, la regla **se descarta**, pero si coinciden **la regla se ejecuta**.

La función `ejecutar_accion()` permite mover mensaje a alguna carpeta antes de agregarlo a la bandeja, permite agregar etiquetas para etiquetar automáticamente mensajes específicos, así como también quitarlas, cambiar la prioridad, y marcarlo como leído

Dejándonos con una secuencia real del proceso de recepción de mensajes tal qué cumplen con las reglas en el siguiente orden:



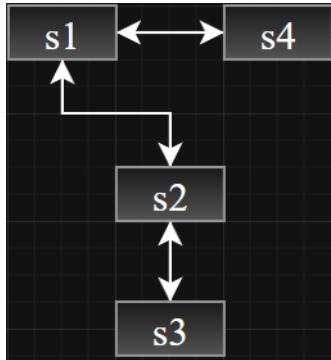
En este caso **Reglas** y **Cola** trabajan juntas, las reglas pueden reordenar, etiquetar y priorizar mensajes ANTES de ingresarlos a la cola.

# Grafo de servidores

La red de servidores se encuentra estructurada de tal forma qué escrita en código es:

```
s1 → s2, s4  
s2 → s1, s3  
s3 → s2  
s4 → s1
```

O expresado de manera visual:



Esto es un **grafo no dirigido**, donde cada servidor es un nodo y las conexiones directas son aristas.

## Algoritmo principal del grafo: camino\_simple() (DFS recursivo)

El algoritmo está definido dentro del módulo servidor\_correo de la siguiente forma:

```
def camino_simple(origen, destino, visit=None):  
    #usando recursividad ve cual es el mejor camino entre los 4 servidores  
    if visit is None:  
        visit = set()  
    if origen == destino:  
        return [destino]  
    visit.add(origen)  
    for v in RED.get(origen, []):  
        if v not in visit:  
            c = camino_simple(v, destino, visit)  
            if c:  
                return [origen] + c  
    return None
```

**camino\_simple()** implementa una **búsqueda en profundidad (DFS)** para encontrar un camino entre dos servidores. Usa recursividad y un conjunto visitado para evitar ciclos evitando que no volvamos a explorar vecinos. Cada llamada explora un servidor vecino y, si encuentra el destino, reconstruye el camino al regresar por la pila de llamadas. Devolviendo la primera ruta válida que haya encontrado.

# Flujo final de Mensajes en el sistema.

Ya teniendo todos los conceptos incluidos en el sistema, nos queda ver cómo queda el ciclo por el cual pasa un mensaje:

