proyecto

Visualización de Comunicación Segura con Cifrado RSA + Vigenère en una Red Geográfica

Y AQUÍ TE EXPLICO LÍNEA POR LÍNEA:

Importación de librerías:

```
# CIFRADO VIGENÈRE EXTENDIDO CON RED SIMULADA

import networkx as nx # Importa la librería networkx para traba
# import matplotlib.pyplot as plt # Importa matplotlib.pyplot pa
# import matplotlib.patches as mpatches # Importa parches de mat
import string # Importa la librería string para manejar cadenas
import random # Importa la librería random para generar números
import hashlib # Importa la librería hashlib para funciones de h
from Crypto.PublicKey import RSA # Importa la clave pública de R
from Crypto.Cipher import PKCS1_OAEP # Importa el cifrado PKCS1_
import binascii # Importa la librería binascii para conversiones
# import matplotlib.animation as animation # Importa la librería
import plotly.graph_objects as go # Importa graph_objects de plo
# import math # Importa la librería math para cálculos matemáti
```

networkx

Se utiliza para crear y manipular la red de dispositivos (representada como un grafo).

matplotlib.pyplot:

Se utiliza para visualizar la red.**string:**

Proporciona acceso a constantes de cadena, en este caso, para definir el alfabeto.**random:**

Se utiliza para seleccionar un nodo espía aleatorio en la simulación MITM y generar la clave simétrica.**hashlib:**

Se utiliza para generar el hash SHA256 para firmar los mensajes.**Crypto.PublicKey.RSA:**

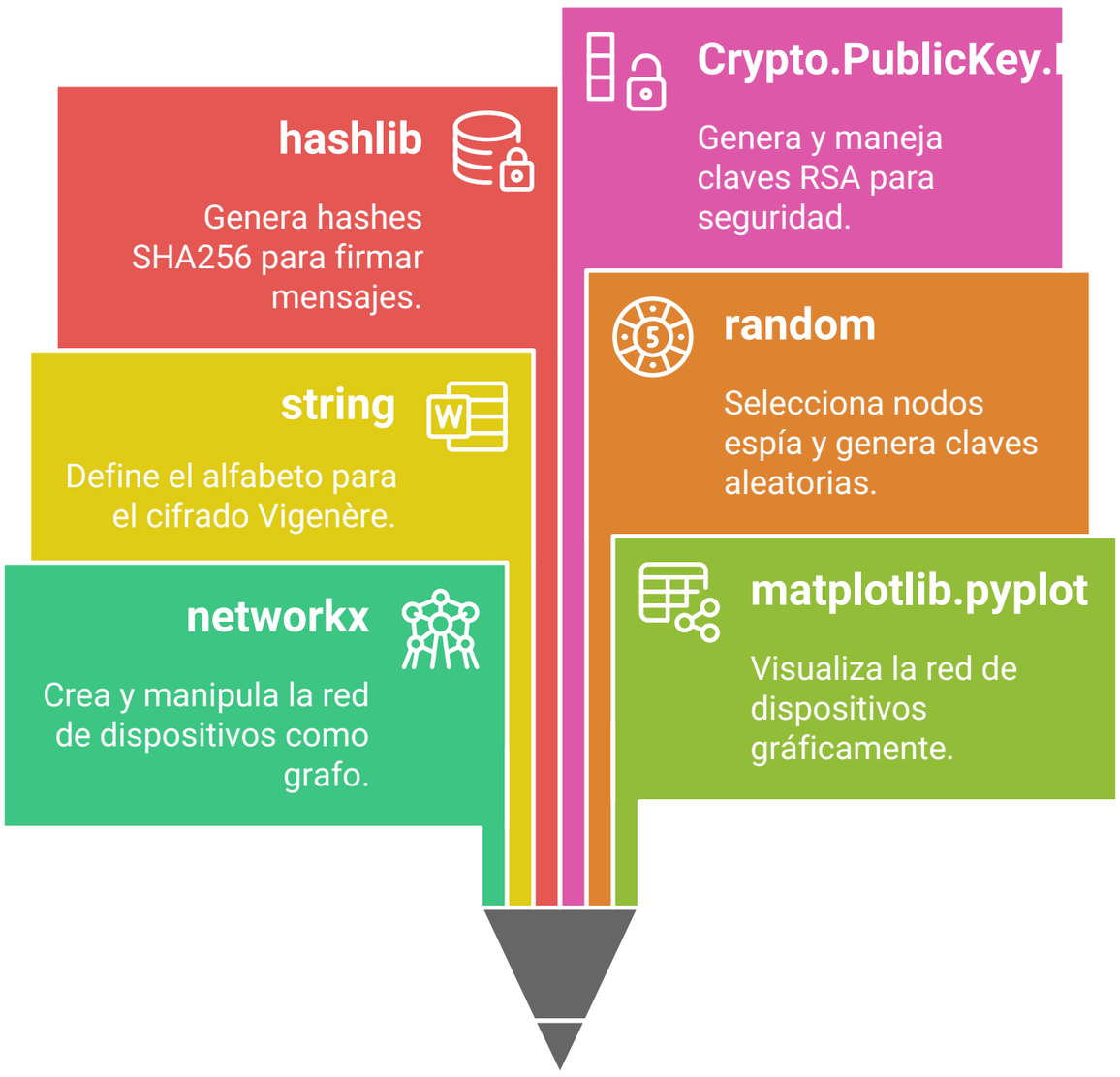
Parte de la librería pycryptodome, se utiliza para generar y manejar las claves RSA (pública y privada).**Crypto.Cipher.PKCS1_OAEP:**

Parte de la librería pycryptodome, se utiliza para cifrar y descifrar datos utilizando el algoritmo RSA con el padding OAEP.**binascii:**

Se utiliza para convertir datos binarios a su representación hexadecimal.**import plotly.graph_objects as go**

Importa graph_objects de plotly para crear gráficos interactivos

LIBRERIAS IMPLEMENTADAS



Definimos el alfabeto que se utilizará para el cifrado Vigenère

```
# Definimos el alfabeto que se utilizará para el cifrado Vigenère
ALFABETO = string.ascii_letters + string.digits + string.punctuation + ' '
```

ALFABETO = string.ascii_letters + string.digits + string.punctuation + ' 'crea una cadena de texto llamada ALFABETO.**string.ascii_letters**representa todas las letras del alfabeto inglés, tanto mayúsculas como minúsculas (abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ).**string.digits**representa todos los dígitos numéricos del 0 al 9 (0123456789).**string.punctuation**representa todos los caracteres de puntuación comunes (!"#$%&'()*+,-./:;<=>?@[]^_{|}~`).' '

es simplemente un espacio en blanco.

Al sumar estas cadenas (**usando el operador +**), se concatenan para formar una única cadena ALFABETO que contiene todas las letras (mayúsculas y minúsculas), todos los dígitos, todos los caracteres de puntuación y un espacio en blanco.Esta cadena se utiliza más adelante en el código como el conjunto de caracteres válidos para el cifrado Vigenère.

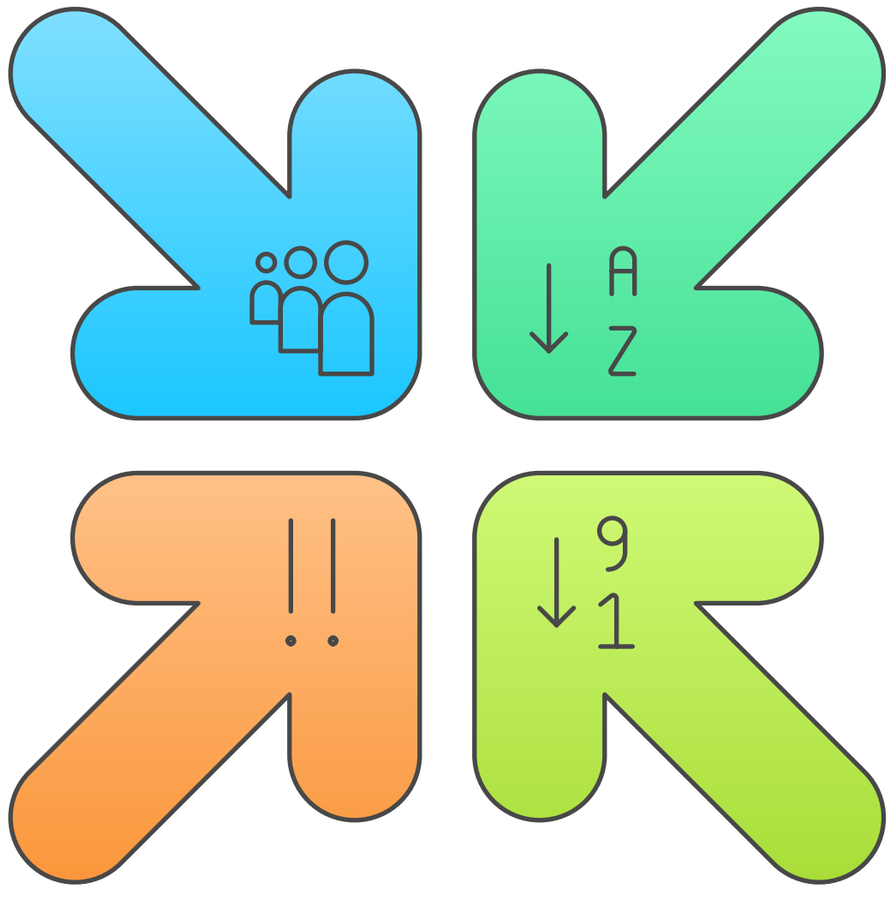
Creación del Alfabeto para el Cifrado Vigenère

Espacio

Proporciona un espacio en blanco para la separación de palabras

Letras ASCII

Incluye todas las letras mayúsculas y minúsculas del alfabeto inglés



Puntuación

Contiene todos los caracteres de puntuación comunes

Dígitos

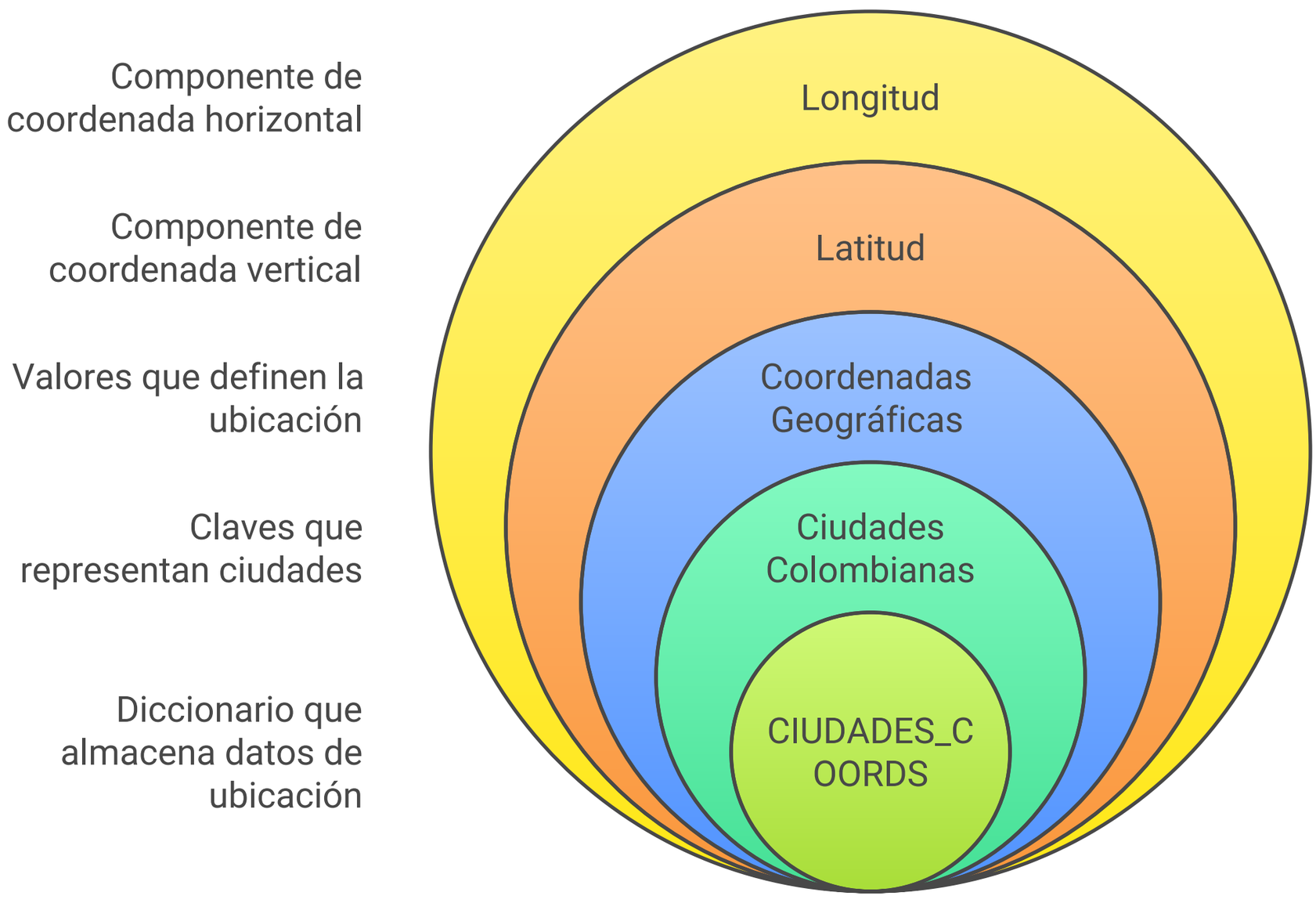
Representa todos los dígitos numéricos del 0 al 9

Coordenadas geográficas de las ciudades colombianas para la visualización

```python
# Coordenadas geográficas de las ciudades colombianas
CIUDADES_COORDS = {
    "Bogotá": (4.711, -74.0721),
    "Medellín": (6.2442, -75.5812),
    "Cali": (3.4516, -76.5319),
    "Barranquilla": (10.9685, -74.7813),
    "Cartagena": (10.3910, -75.4794),
    "Bucaramanga": (7.1193, -73.1227),
    "Cúcuta": (7.8939, -72.5078),
    "Pereira": (4.8087, -75.6906),
    "Manizales": (5.0689, -75.5174),
    "Ibagué": (4.4389, -75.2322),
    "Pasto": (1.2136, -77.2811),
    "Neiva": (2.9965, -75.2908),
    "Villavicencio": (4.1438, -73.6224),
    "Santa Marta": (11.2408, -74.2048),
    "Montería": (8.7609, -75.8822),
    "Armenia": (4.5388, -75.6816),
    "Sincelejo": (9.3047, -75.3973),
    "Popayán": (2.4410, -76.6029),
    "Tunja": (5.5350, -73.3677),
    "Riohacha": (11.5444, -72.9078),
    "Valledupar": (10.4634, -73.2934),
    "Apartadó": (7.8834, -76.6117),
    "Florencia": (1.6137, -75.6060),
    "Quibdó": (5.6932, -76.6585),
    "Arauca": (7.0841, -70.7611),
    "San Andrés": (12.5833, -81.7000),
    "Leticia": (-4.2167, -69.9333)
}
```

CIUDADES_COORDS. define un diccionarioUn diccionario en Python es una colección de pares clave-valor.En este caso, las claves son los nombres de algunas ciudades colombianas (como "Bogotá", "Medellín", "Cali", etc.).Los valores asociados a cada ciudad son tuplas (pares de números).

Estos números representan las coordenadas geográficas de cada ciudad: el primer número es la latitud y el segundo número es la longitud.Este diccionario se utiliza en el código para asignar una ubicación geográfica a cada nodo de la red simulada, permitiendo visualizarlos en un mapa.



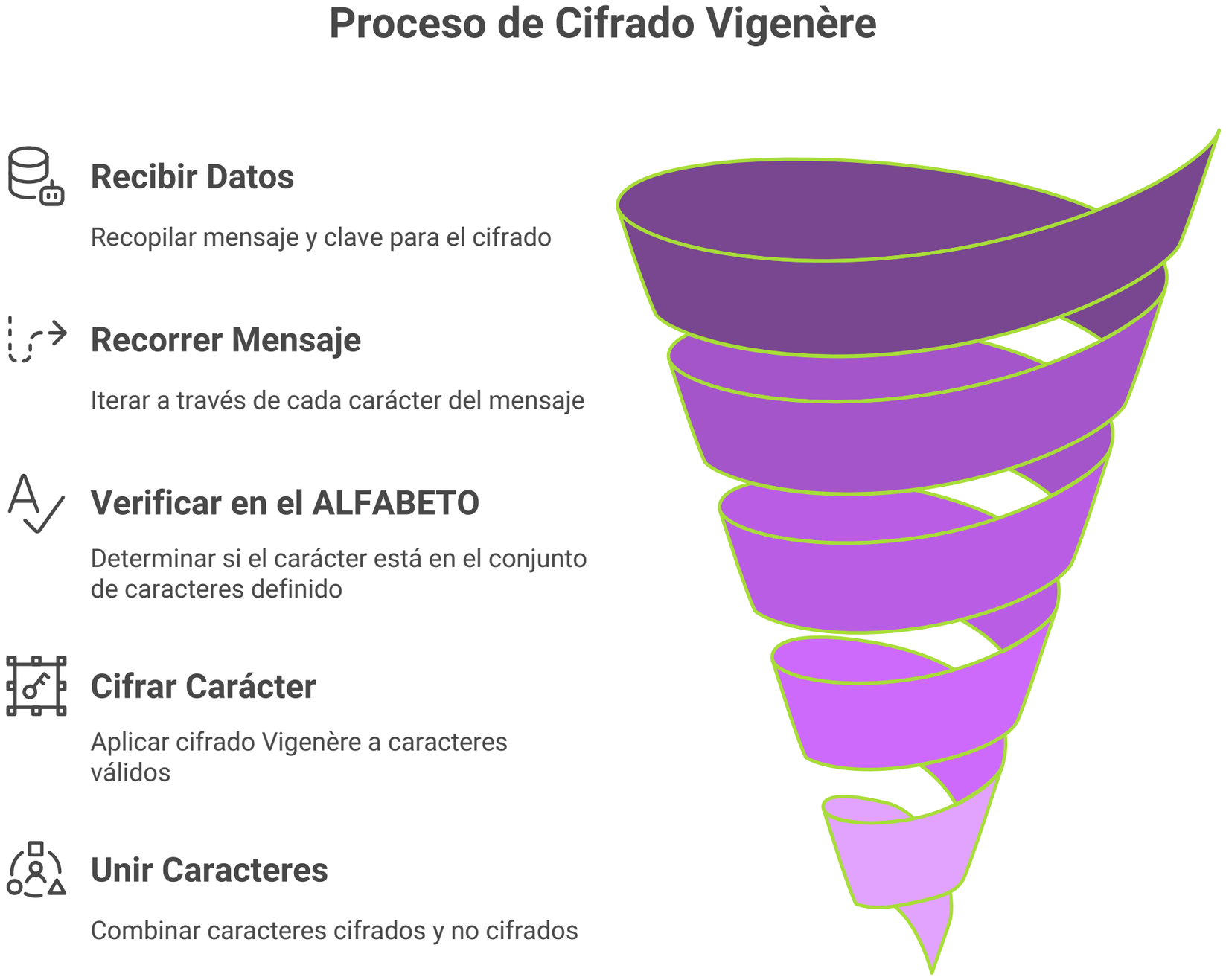
Esta función se encarga de cifrar un mensaje de texto utilizando el método de cifrado Vigenère.
Aquí te detallo qué hace:

```
def vigenere_cifrar(mensaje, clave):
    """
    Cifra un mensaje usando el cifrado Vigenère con una clave dada.

    Toma un mensaje y una clave como entrada. Itera sobre cada carácter del mensaje y,
    si el carácter se encuentra en el ALFABETO definido, lo cifra utilizando la clave
    y la posición del carácter en el alfabeto. Los caracteres que no están en el
    alfabeto se dejan sin cifrar.

    Args:
        mensaje (str): El mensaje a cifrar.
        clave (str): La clave para el cifrado Vigenère.

    Returns:
        str: El mensaje cifrado.
    """
```

1. **Recibe dos datos:**

1. El mensaje que quieres cifrar y la clave que usarás para el cifrado.
2. **Recorre el mensaje:**
2. La función va revisando cada letra o carácter del mensaje uno por uno.
3. **Verifica si está en el ALFABETO:**
3. Para cada carácter del mensaje, comprueba si ese carácter se encuentra dentro de la cadena ALFABETO que definimos anteriormente (la que contiene letras, números, puntuación y el espacio).
4. **Cifra el carácter (si está en el ALFABETO):**
4. Si el carácter del mensaje está en el ALFABETO, la función realiza el cifrado Vigenère. Esto implica:Encontrar la posición del carácter actual en el ALFABETO.Encontrar la posición del carácter correspondiente en la clave (la clave se repite si es más corta que el mensaje).Sumar estas dos posiciones.Calcular el módulo de esta suma respecto a la longitud del ALFABETO (esto asegura que el resultado esté dentro del rango del ALFABETO).El carácter en el ALFABETO que se encuentra en la posición resultante es el carácter cifrado.

5. **Deja el carácter sin cifrar (si NO está en el ALFABETO):**
5. Si un carácter del mensaje no se encuentra en el ALFABETO (por ejemplo, si fuera un carácter especial que no incluimos), la función lo deja tal cual, sin modificarlo.
6. **Une los caracteres cifrados:**
6. Una vez que ha procesado todos los caracteres del mensaje, une los caracteres cifrados (y los no cifrados) para formar la cadena final.
7. **Devuelve el mensaje cifrado:**
La función retorna esta cadena que es el mensaje original pero ya cifrado.



En resumen, vigenere_cifrar aplica el cifrado Vigenère solo a los caracteres definidos en ALFABETO, usando la clave proporcionada.

Esta sección es una list comprensión (comprensión de lista) combinada con '.'.join(), que es una forma compacta y eficiente en Python de construir una nueva cadena de texto.Vamos a desglosarla:

```
# Itera sobre cada carácter del mensaje con su índice
return ''.join(
    # Cifra el carácter si está en el alfabeto
    ALFABETO[(ALFABETO.index(c) + ALFABETO.index(clave[i % len(clave)])) % len(ALFABETO)]
    if c in ALFABETO else c # Deja el carácter sin cifrar si no está en el alfabeto
    for i, c in enumerate(mensaje)
)
```

for i, c in enumerate(mensaje): Esta parte itera sobre cada carácter (c) del mensaje de entrada. enumerate también proporciona el índice (i) de cada carácter a medida que itera. Esto es crucial para el cifrado Vigenère, ya que la clave se aplica repetidamente según el índice del carácter del mensaje.**if c in ALFABETO else c:**Esta es una expresión condicional. Para cada carácter c del mensaje:**if c in ALFABETO:**Comprueba si el carácter c se encuentra en nuestra cadena ALFABETO.**... ALFABETO[...] ...:**Si el carácter c SÍ está en el ALFABETO, entonces se ejecuta la parte del cifrado Vigenère que está dentro de los corchetes ALFABETO[...].**else c:**Si el carácter c NO está en el ALFABETO, simplemente se mantiene el carácter original c sin modificarlo.**ALFABETO[(ALFABETO.index(c) + ALFABETO.index(clave[i % len(clave)])) % len(ALFABETO)]:**Esta es la parte central del cifrado Vigenère que se ejecuta solo si el carácter c está en el

ALFABETO. Vamos a desglosar esto también:

ALFABETO.index(c): Encuentra la posición (índice numérico) del carácter actual c dentro de la cadena ALFABETO.**clave[i % len(clave)]:**Selecciona el carácter de la clave que se va a usar para cifrar el carácter actual c. El operador % (módulo) se asegura de que si el índice i del mensaje es mayor que la longitud de la clave, la clave se "envuelva" y se use desde el principio nuevamente. Por ejemplo, si la clave tiene 5 letras y estamos en el carácter 7 del mensaje, se usaría el carácter de la clave en la posición 7 % 5 = 2.**ALFABETO.index(clave[i % len(clave)]):**Encuentra la posición (índice numérico) del carácter de la clave seleccionado en el paso anterior, dentro de la cadena ALFABETO.**(ALFABETO.index(c) + ALFABETO.index(clave[i % len(clave)])):**Suma las posiciones del carácter del mensaje y del carácter de la clave en el ALFABETO.**... % len(ALFABETO):**Calcula el módulo de la suma anterior respecto a la longitud total del ALFABETO. Esto es importante porque si la suma de las posiciones excede el número de caracteres en el ALFABETO, el módulo "envuelve" el resultado de vuelta al inicio del ALFABETO.**ALFABETO[...]:**Finalmente, usa el resultado del módulo como un índice para seleccionar el carácter cifrado de la cadena ALFABETO.
''.join(...): Toda la lógica anterior (el for y el if/else) genera una secuencia de caracteres (los caracteres cifrados o los originales si no estaban en el ALFABETO). El método join('') toma esta secuencia de caracteres y los une todos juntos para formar una única cadena de texto. El '' antes de .join() indica que no se coloque ningún separador entre los caracteres unidos.

En resumen, esta línea de código genera el mensaje cifrado iterando sobre el mensaje original, cifrando cada carácter si está en el alfabeto definido y dejando los demás sin modificar, y luego uniendo todos estos caracteres procesados en una sola cadena que es el resultado del cifrado

Esta función simula una firma digital simple para un mensaje.

Aquí te desglosamos lo que hace:

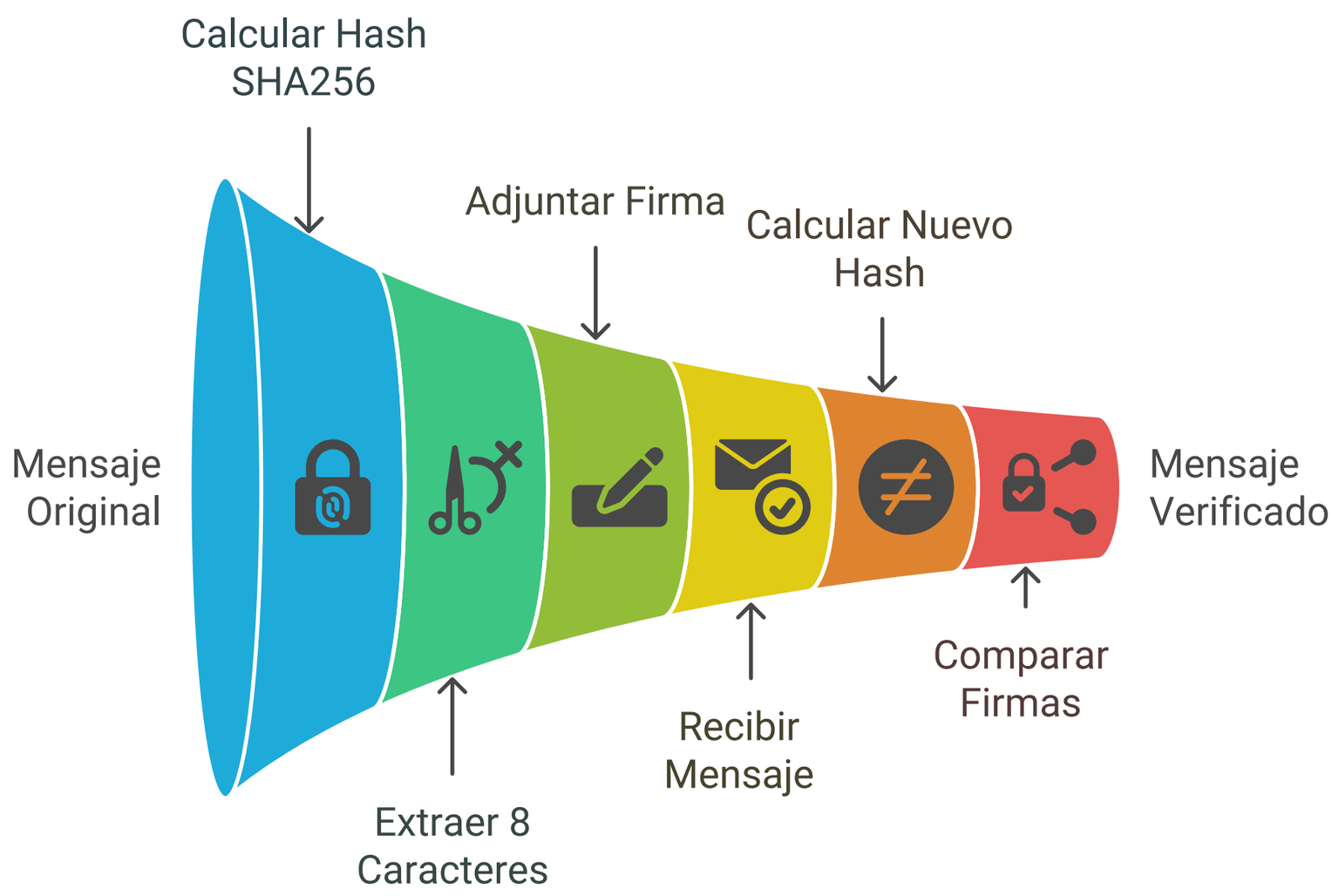
```python
def firmar_mensaje(mensaje):
    """
    Firma digitalmente un mensaje calculando su hash SHA256 y adjuntando los primeros 8

    Simula una firma digital simple. Toma un mensaje, calcula su hash SHA256 y luego
    toma los primeros 8 caracteres de ese hash. Devuelve el mensaje original
    concatenado con "||" y estos 8 caracteres del hash como la "firma". Esto permite
    verificar si el mensaje fue alterado posteriormente (aunque no proporciona
    autenticación real como una firma digital con claves privadas/públicas).

    Args:
        mensaje (str): El mensaje a firmar.
    Returns:
        str: El mensaje firmado en formato "mensaje||firma".
    """
    # Calcula el hash SHA256 del mensaje
    hash_completo = hashlib.sha256(mensaje.encode()).hexdigest()
    # Toma los primeros 8 caracteres del hash
    hash8 = hash_completo[:8]
    # Concatena el mensaje original con "||" y los 8 caracteres del hash
    return f"{mensaje}||{hash8}"
```

1. **def firmar_mensaje(mensaje):**

   Define una función llamada firmar_mensaje que toma un argumento, mensaje, que es la cadena de texto que queremos "firmar".

2. **Docstring (""" ... """):**
2. Esta sección explica qué hace la función, sus argumentos (Args) y lo que devuelve (Returns). Indica que la función calcula un hash SHA256 del mensaje y usa los primeros 8 caracteres como una "firma" simple. También aclara que esto es solo una simulación y no una firma digital real con claves criptográficas.
3. **hash_completo = hashlib.sha256(mensaje.encode()).hexdigest():**
    - **mensaje.encode():** Convierte la cadena de texto del mensaje en bytes, ya que las funciones hash trabajan con bytes.
    - **hashlib.sha256(...):** Calcula el hash SHA256 de los bytes del mensaje. SHA256 es un algoritmo de hashing criptográfico que produce una cadena de longitud fija (64 caracteres hexadecimales) que es única para cada mensaje de entrada. Cambiar incluso un solo carácter en el mensaje original resultaría en un hash completamente diferente.
    - .**hexdigest():** Convierte el hash resultante (que está en formato binario) en una cadena hexadecimal más legible.
4. **hash8 = hash_completo[:8]:**
4. Toma la cadena hexadecimal completa del hash (hash_completo) y selecciona solo los primeros 8 caracteres utilizando slicing de cadenas. Esta es la parte que se usará como la "firma" simulada.
5. **return f"{mensaje}||{hash8}":**
5. Construye y devuelve una nueva cadena de texto. Esta cadena está formada por el mensaje original, seguido de dos barras verticales (||), y finalmente los 8 caracteres del hash (hash8). Este formato (mensaje||firma) permite separar el mensaje de la firma más adelante para su verificación.

Proceso de Verificación de Mensajes



En resumen, la función toma tu mensaje, calcula una especie de huella digital única (el hash SHA256), toma un pedacito de esa huella (los primeros 8 caracteres) y lo adjunta al final de tu mensaje, separado por "||". Esto te permite, en teoría, verificar si el mensaje ha sido modificado después de ser "firmado", comparando la "firma" adjunta con una nueva huella digital calculada sobre el mensaje recibido (como se hace en la función verificar_firma).

Esta función se encarga de verificar si un mensaje que supuestamente ha sido "firmado" (usando la función firmar_mensaje **que vimos antes) realmente no ha sido modificado desde que se le adjuntó la "firma".**

Aquí tienes el detalle de lo que hace:

```python
def verificar_firma(mensaje_firmado):
    """
    Verifica la firma digital de un mensaje firmado.

    Verifica la "firma" de un mensaje que fue firmado previamente con firmar_mensaje.
    Toma el mensaje_firmado en formato "mensaje||firma", lo divide en el mensaje
    original y la firma. Recalcula el hash SHA256 del mensaje original extraído y
    compara los primeros 8 caracteres con la firma proporcionada.

    Args:
        mensaje_firmado (str): El mensaje firmado en formato "mensaje||firma".

    Returns:
        tuple: Una tupla que contiene un booleano (True si la firma es válida, False si
               y el mensaje original extraído.
    """
    try:
        # Divide el mensaje firmado en mensaje original y firma
        mensaje, firma = mensaje_firmado.rsplit('||', 1)
        # Recalcula el hash del mensaje original extraído y compara los primeros 8 carac
        return firma == hashlib.sha256(mensaje.encode()).hexdigest()[:8], mensaje
    except ValueError:
        # Si no se puede dividir el mensaje (formato incorrecto), la firma es inválida
        return False, mensaje_firmado
```

def verificar_firma(mensaje_firmado):: Define una función llamada verificar_firma que toma un argumento, mensaje_firmado. Se espera que este argumento sea una cadena de texto en el formato "mensaje original || firma" que fue producido por la función firmar_mensaje.
Docstring (""" ... """): Explica el propósito de la función: verificar la "firma" de un mensaje firmado previamente. Describe cómo espera el formato de entrada (mensaje_firmado) y qué devuelve (Returns): una tupla con un valor booleano (True si la firma es correcta, False si no) y el mensaje original extraído.
try...except ValueError:: Este bloque try intenta ejecutar el código que podría generar un error. Si ocurre un ValueError (por ejemplo, si la cadena mensaje_firmado no contiene || y no se puede dividir), el código dentro del bloque except se ejecuta para manejar ese error.
Dentro del try:
mensaje, firma = mensaje_firmado.rsplit('||', 1): Esta es la parte clave para separar el mensaje de la firma.
mensaje_firmado.rsplit('||', 1): Divide la cadena mensaje_firmado en una lista de subcadenas, usando || como separador. El 1 al final le indica a Python que solo haga una división, comenzando desde el final de la cadena. Esto es importante por si el mensaje original contiene ||. Así, siempre separará por el *último* ||, asumiendo que la "firma" es lo que viene después de él.
mensaje, firma = ...: Asigna los elementos de la lista resultante a dos variables: mensaje (la parte antes del último ||) y firma (la parte después del último ||).
return firma == hashlib.sha256(mensaje.encode()).hexdigest()[:8], mensaje: Aquí es donde se realiza la verificación.
hashlib.sha256(mensaje.encode()).hexdigest()[:8]: Calcula el hash SHA256 del mensaje que acaba de extraer (primero lo codifica a bytes con .encode()) y toma los primeros 8 caracteres de ese nuevo hash. Este es el hash que *debería* coincidir con la firma que venía adjunta.
firma == ...: Compara la firma extraída del mensaje_firmado con los primeros 8 caracteres del hash recién calculado del mensaje extraído. Esta comparación devuelve True si son iguales (lo que significa que el mensaje no ha sido alterado) o False si son diferentes.
, mensaje: La función devuelve una tupla que contiene el resultado de la comparación (True o False) y el mensaje original que se extrajo.
Dentro del except ValueError:

return False, mensaje_firmado: Si ocurre un ValueError durante el rsplit (porque el formato no es el esperado), la función devuelve False (indicando que la firma no es válida, ya que el formato es incorrecto) y devuelve la cadena mensaje_firmado original tal como la recibió.

¿Cómo manejar la verificación de la firma en el procesamiento de mensajes?

Verificación Exitosa

Asegura la integridad del mensaje y la autenticidad

Verificación Fallida

Indica posible manipulación o formato incorrecto

En resumen, esta función intenta separar el mensaje de la firma, recalcula la "huella digital" del mensaje extraído y la compara con la "huella digital" que venía adjunta. Si coinciden, considera que la firma es válida y el mensaje no fue alterado. Si no coinciden o el formato es incorrecto, considera que la firma no es válida.

Esta función se encarga de construir la red simulada de dispositivos (nodos) que se utilizará en el programa. Piensa en ella como la creadora del mapa y los puntos de conexión.

Aquí te detallo cada parte:

```python
def crear_red_dispositivos(n):
    """
    Crea una red aleatoria de dispositivos (nodos) distribuidos en ciudades colombianas.

    A cada nodo se le asigna un par de claves RSA (pública y privada) y una ubicación
    geográfica. Crea un grafo aleatorio usando el modelo de Erdos-Renyi y asegura
    que sea conexo (que todos los nodos estén conectados).

    Args:
        n (int): El número de nodos (dispositivos) a crear.

    Returns:
        nx.Graph: El grafo que representa la red de dispositivos.
    """
    # Obtiene la lista de ciudades y la mezcla aleatoriamente
    ciudades = list(CIUDADES_COORDS.keys())
    random.shuffle(ciudades)

    # Crea un grafo aleatorio usando el modelo de Erdos-Renyi
    G = nx.erdos_renyi_graph(n, 0.4)
    # Asegura que el grafo sea conexo (todos los nodos conectados)
    while not nx.is_connected(G):
        G = nx.erdos_renyi_graph(n, 0.4)
```

def crear_red_dispositivos(n):: Define la función crear_red_dispositivos que toma un único argumento, n. Este n representa el número de dispositivos (nodos) que quieres que tenga la red.
Docstring (""" ... """): Describe el propósito de la función: crear una red aleatoria de dispositivos ubicados en ciudades colombianas. Explica que a cada dispositivo se le asignan claves de cifrado (RSA) y una ubicación. También menciona que utiliza un modelo de grafo aleatorio (Erdos-Renyi) y se asegura de que la red esté completamente conectada (conexo).
ciudades = list(CIUDADES_COORDS.keys()):

- Accede al diccionario **CIUDADES_COORDS** (que definimos al inicio del código y contiene las ciudades y sus coordenadas).

- **.keys():** Obtiene una vista de todas las claves (los nombres de las ciudades) de ese diccionario.
- **list(...):** Convierte esa vista de claves en una lista real de nombres de ciudades.

random.shuffle(ciudades): Mezcla aleatoriamente el orden de los nombres de las ciudades en la lista ciudades. Esto se hace para que, al asignar ciudades a los nodos más adelante, la distribución de nodos por ciudad sea más variada.
G = nx.erdos_renyi_graph(n, 0.4):

- nx.erdos_renyi_graph(n, 0.4): Esta es la parte clave donde se crea el grafo (la red). Utiliza el modelo de Erdos-Renyi de la librería networkx.
  - **n:** Es el número de nodos que tendrá el grafo (el número de dispositivos que especificaste al llamar la función).
  - **0.4:** Es la probabilidad p. En el modelo de Erdos-Renyi, esto significa que por cada posible par de nodos en el grafo, hay una probabilidad del 40% (0.4) de que exista una conexión (una arista) entre ellos. Un valor más alto de p crea una red más densamente conectada.

while not nx.is_connected(G): G = nx.erdos_renyi_graph(n, 0.4):

- **nx.is_connected(G):** Esta función de networkx verifica si el grafo G es "conexo". Un grafo conexo es aquel en el que hay un camino entre cualquier par de nodos. En el contexto de la red simulada, significa que cualquier dispositivo puede comunicarse (a través de otros dispositivos si es necesario) con cualquier otro dispositivo.
- **while not ...:** Este bucle while continúa ejecutándose *mientras* el grafo G *no* sea conexo.
- **G = nx.erdos_renyi_graph(n, 0.4):** Si el grafo generado en el paso anterior no era conexo, se genera un *nuevo* grafo aleatorio de Erdos-Renyi con el mismo número de nodos (n) y la misma probabilidad (0.4). El bucle sigue generando grafos aleatorios hasta que uno de ellos resulte ser conexo. Esto asegura que todos los dispositivos en tu red puedan, teóricamente, enviarse mensajes entre sí.

Después de este bucle, la función continúa asignando las claves RSA y las ubicaciones de las ciudades a cada nodo del grafo G, como se describe en el resto del código de la función (que no incluiste en esta solicitud, pero está presente en tu cuaderno)

Creación de Red de Dispositivos

Definir Función

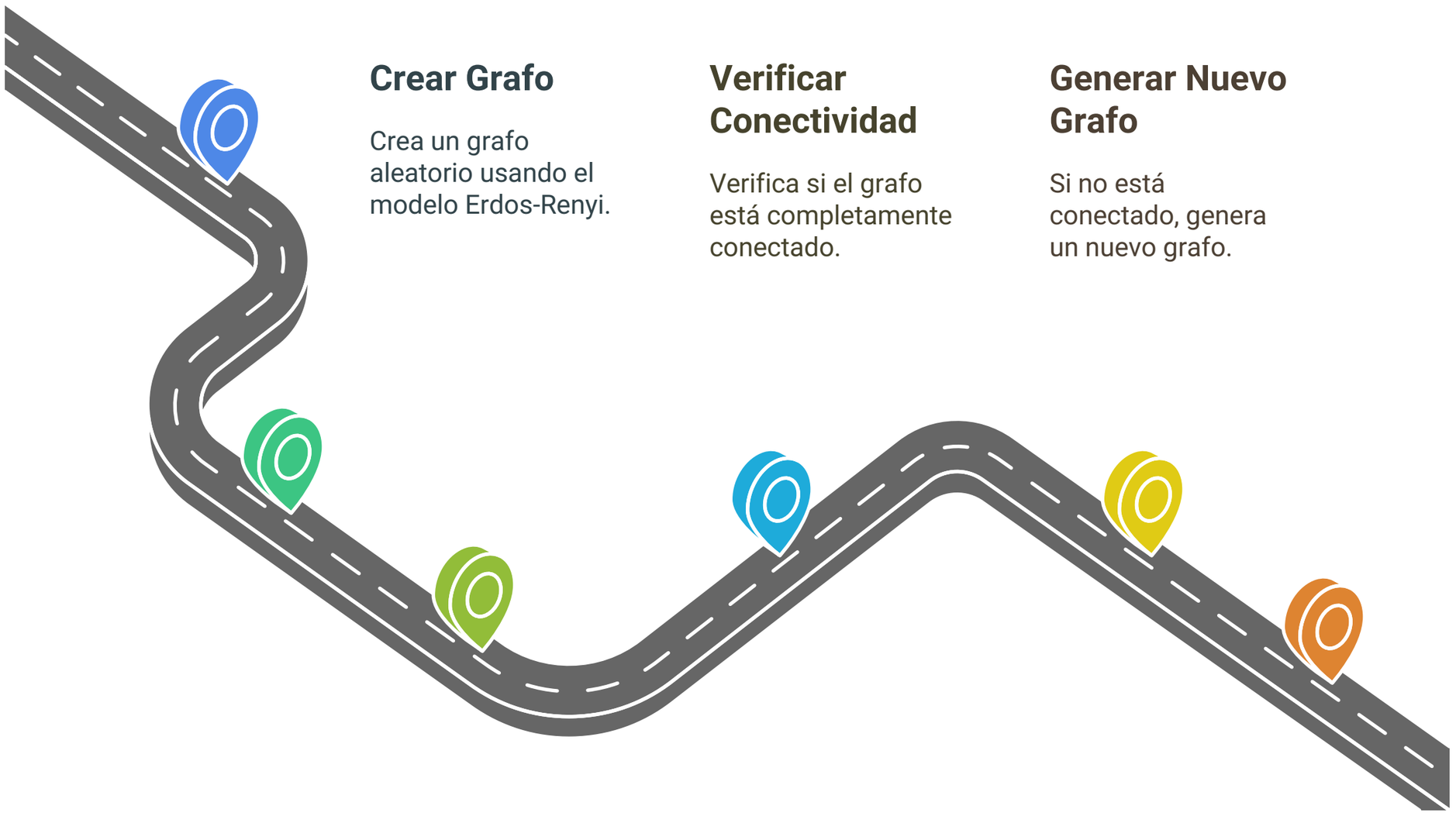
Define la función para crear la red.

Acceder a Nombres de Ciudades

Obtiene una lista de nombres de ciudades.

Mezclar Ciudades

Mezcla aleatoriamente el orden de las ciudades.



Crear Grafo

Crea un grafo aleatorio usando el modelo Erdos-Renyi.

Verificar Conectividad

Verifica si el grafo está completamente conectado.

Generar Nuevo Grafo

Si no está conectado, genera un nuevo grafo.

En resumen, esta función crea la estructura de la red de dispositivos, asegurándose de que haya suficientes conexiones aleatorias para que todos los dispositivos puedan comunicarse, y les asigna identidades criptográficas y ubicaciones geográficas.

Continuamos con la explicación de la última parte de la función crear_red_dispositivos**.**

```
# Asigna claves RSA y ubicación a cada nodo
for nodo in G.nodes:
    # Genera un par de claves RSA (pública y privada)
    key = RSA.generate(2048)
    G.nodes[nodo]['rsa_private'] = key
    G.nodes[nodo]['rsa_public'] = key.publickey()
    # Asigna una ciudad al nodo
    ciudad = ciudades[nodo % len(ciudades)]
    G.nodes[nodo]['ciudad'] = ciudad
    G.nodes[nodo]['pos'] = CIUDADES_COORDS[ciudad] # Asigna coordenadas de la ciudad
return G
```

Después de crear un grafo conexo con n nodos usando el modelo de Erdos-Renyi (como explicamos antes), este bloque de código recorre cada uno de esos nodos y les asigna propiedades importantes: un par de claves RSA para cifrado asimétrico y una ubicación geográfica en una ciudad colombiana.

Aquí tienes el desglose:

1. **for nodo in G.nodes:**: Este bucle for itera sobre cada nodo individual en el grafo G que acabamos de crear y asegurar que sea conexo. Para cada iteración, la variable nodo toma el valor del índice del nodo actual (por ejemplo, 0, 1, 2, hasta n-1).
2. **key = RSA.generate(2048)**:
    - **RSA.generate(2048):** Utiliza la biblioteca PyCryptodome (importada como RSA al principio del script) para generar un nuevo par de claves RSA. El número 2048 especifica la longitud de la clave en bits. Generalmente, una longitud mayor (como 2048 o 4096 bits) proporciona una mayor seguridad. Este proceso crea dos partes: una clave privada secreta y una clave pública que puede ser compartida.
3. **G.nodes[nodo]['rsa_private'] = key**:
    - **G.nodes[nodo]:** Accede a los atributos del nodo actual en el grafo G. Los grafos en networkx permiten almacenar información adicional (atributos) con cada nodo.
    - **['rsa_private'] = key:** Crea un nuevo atributo llamado 'rsa_private' para el nodo actual y le asigna la clave privada RSA que se acaba de generar (key). Esta clave privada es secreta y solo el propietario del nodo debe tener acceso a ella.
4. **G.nodes[nodo]['rsa_public'] = key.publickey()**:
    - **key.publickey():** Obtiene la parte de clave pública del par de claves RSA que se generó.
    - **G.nodes[nodo]['rsa_public'] = ...:** Crea un nuevo atributo llamado 'rsa_public' para el nodo actual y le asigna la clave pública correspondiente. Esta clave pública se usará para cifrar mensajes *para* este nodo o para verificar firmas *de* este nodo.
5. **ciudad = ciudades[nodo % len(ciudades)]**:
    - **ciudades:** Es la lista de ciudades colombianas que mezclamos aleatoriamente al principio de la función.
    - l**en(ciudades):** Obtiene el número total de ciudades en la lista.
    - **nodo % len(ciudades):** Utiliza el operador módulo (%). Esto calcula el resto de la división del índice del nodo (nodo) por el número total de ciudades. Esto asegura que, a medida que iteramos por los nodos (0, 1, 2, ...), asignemos ciudades de la lista de forma cíclica (ciudad 0, ciudad 1, ..., última ciudad, ciudad 0, ciudad 1, ...). De esta manera, distribuimos las ciudades disponibles entre todos los nodos.

Asegurando Nodos de Red con Cifrado RSA

En resumen, este segmento de código personaliza cada dispositivo (nodo) en la red generada, dándole las herramientas criptográficas necesarias para una comunicación segura (claves RSA) y una ubicación en el mapa simulado

Ahora vamos a explicar la función enviar_mensaje_unico**, que es el corazón de la simulación, ya que modela cómo un mensaje viaja a través de la red, cómo se cifra y descifra, y cómo un espía podría intervenir.**
Aquí tienes el detalle:

```python
def enviar_mensaje_unico(G, origen, destino, mensaje, espia_activo=True):
    """
    Simula el envío de un mensaje cifrado y firmado entre dos nodos de la red.

    Utiliza cifrado Vigenère para el mensaje y cifrado RSA para la clave simétrica.
    Opcionalmente, simula un ataque Man-in-the-Middle (MITM) donde un nodo espía
    intercepta el mensaje.

    Args:
        G (nx.Graph): El grafo que representa la red de dispositivos.
        origen (int): El índice del nodo de origen.
        destino (int): El índice del nodo de destino.
        mensaje (str): El mensaje a enviar.
        espia_activo (bool, optional): True para activar la simulación del espía, False
                                       Por defecto es True.

    Returns:
        tuple: Una tupla que contiene el camino recorrido por el mensaje y el nodo espía
    """
```

2. **def enviar_mensaje_unico(G, origen, destino, mensaje, espia_activo=True):**: Define la función enviar_mensaje_unico con varios argumentos:
    - **G:** El grafo (nx.Graph) que representa la red de dispositivos, con todos los atributos que le añadimos antes (claves RSA, ciudades, posiciones).
    - **origen:** El índice (número) del nodo que envía el mensaje.
    - **destino:** El índice (número) del nodo que debe recibir el mensaje.
    - **mensaje:** La cadena de texto que se quiere enviar.
    - **espia_activo:** Un argumento booleano opcional que, si es True, activa la simulación de un nodo espía interceptando el mensaje. Por defecto es True.
3. **Docstring (""" ... """)**: Describe la función: simula el envío de un mensaje cifrado y firmado usando Vigenère y RSA. Menciona la simulación opcional del ataque MITM (Man-in-the-Middle) y explica los argumentos y lo que devuelve (Returns): una tupla con la ruta que siguió el mensaje y el nodo espía (si hubo uno).

```python
    # Calcula el camino más corto entre origen y destino
    camino = nx.shortest_path(G, origen, destino)
    # Selecciona un nodo espía aleatorio en el camino (excluyendo origen y destino) si e
    espia = random.choice(camino[1:-1]) if espia_activo and len(camino) > 2 else None

    # Genera una clave simétrica aleatoria para el cifrado Vigenère
    clave_simetrica = ''.join(random.choice(ALFABETO) for _ in range(16))
    # Firma digitalmente el mensaje usando SHA256
    mensaje_firmado = firmar_mensaje(mensaje)
    # Cifra el mensaje firmado usando Vigenère con la clave simétrica
    mensaje_cifrado = vigenere_cifrar(mensaje_firmado, clave_simetrica)

    # Obtiene la clave pública del nodo destino
    public_key = G.nodes[destino]['rsa_public']
    # Crea un objeto de cifrado RSA con PKCS1_OAEP
    cipher_rsa = PKCS1_OAEP.new(public_key)
    # Cifra la clave simétrica usando la clave pública del destino
    clave_cifrada = cipher_rsa.encrypt(clave_simetrica.encode())

    # Imprime información sobre el envío del mensaje
    print(f"\n📤 Enviando mensaje de nodo {origen} a nodo {destino}:")
    print(" - Camino:", [f"{n} ({G.nodes[n]['ciudad']})" for n in camino])
    print(" - Clave cifrada (hex):", binascii.hexlify(clave_cifrada).decode())
    print(" - Mensaje cifrado:", mensaje_cifrado)
```

camino = nx.shortest_path(G, origen, destino):

- **nx.shortest_path(G, origen, destino):** Utiliza networkx para calcular el camino más corto (en términos de número de nodos) entre el nodo origen y el nodo destino en el grafo G. Devuelve una lista de los nodos por los que pasaría el mensaje.

espia = random.choice(camino[1:-1]) if espia_activo and len(camino) > 2 else None:

- Esta línea es una expresión condicional **(if ... else)** que decide si habrá un nodo espía y cuál será.
- **espia_activo and len(camino) > 2:** La condición para que haya un espía es doble: que espia_activo sea True *y* que el camino tenga más de 2 nodos (es decir, que no sea una conexión directa entre origen y destino, ya que el espía se coloca *entre* ellos).

- **random.choice(camino[1:-1]):** Si la condición es True, selecciona un nodo aleatorio de la lista camino, excluyendo el primer nodo (camino[0], el origen) y el último nodo (camino[-1], el destino) usando el slicing [1:-1]. Este nodo seleccionado será el espia.
- **else None:** Si la condición es False (o el camino es muy corto), la variable espia se establece en None, indicando que no hay espía activo en esta simulación.

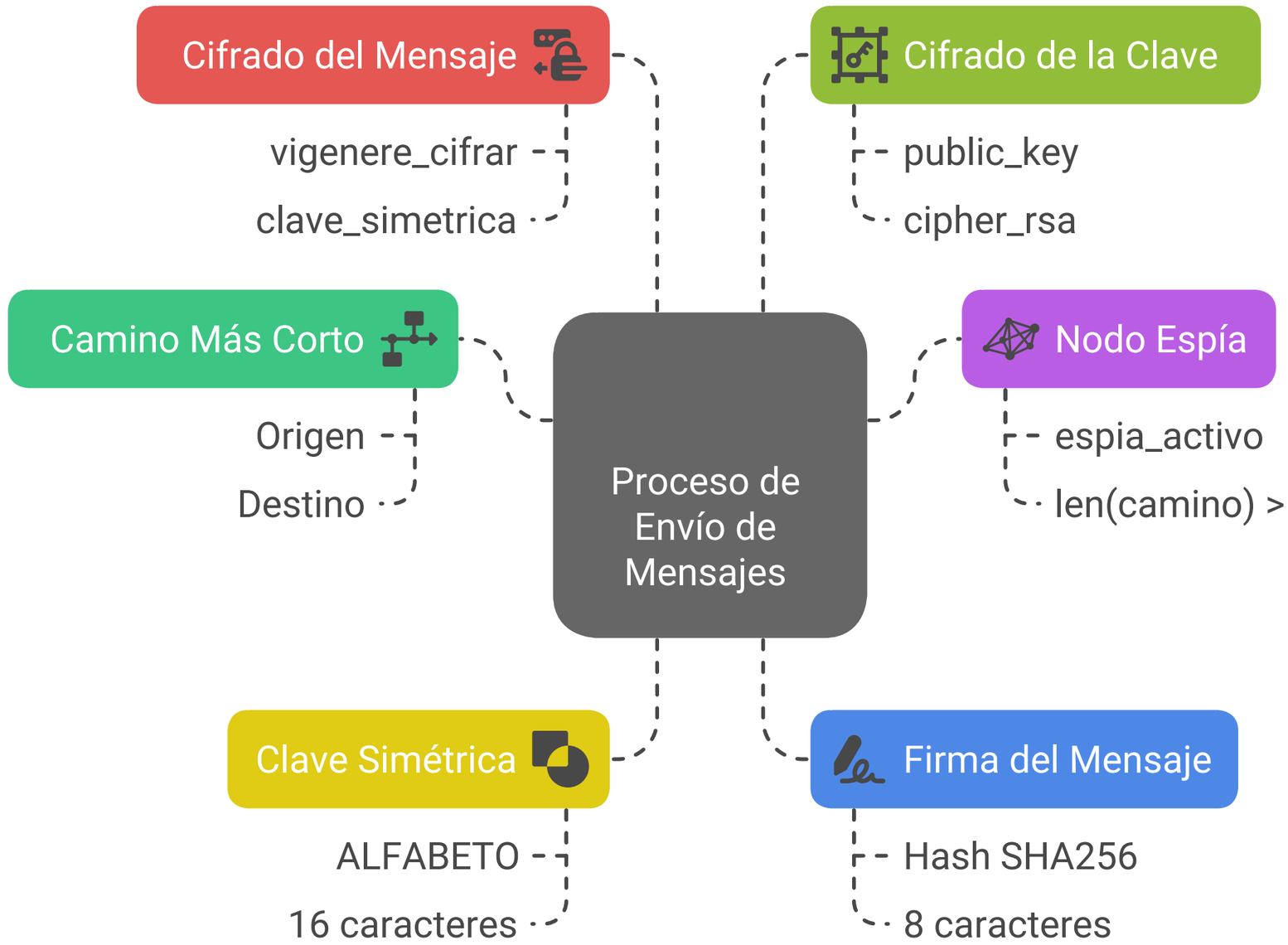
clave_simetrica = ''.join(random.choice(ALFABETO) for _ in range(16)):

- **random.choice(ALFABETO):** Selecciona un carácter aleatorio de la lista ALFABETO (que contiene letras, números, puntuación y espacio).
- **for _ in range(16):** Repite la selección aleatoria 16 veces.
- **''.join(...):** Une los 16 caracteres aleatorios seleccionados en una sola cadena. Esta cadena de 16 caracteres será la clave simétrica que se usará para el cifrado y descifrado Vigenère. Se genera una clave nueva para cada mensaje enviado.

mensaje_firmado = firmar_mensaje(mensaje): Llama a la función firmar_mensaje (que explicamos antes) para tomar el mensaje original, calcular su hash SHA256, tomar los primeros 8 caracteres y adjuntarlos al mensaje en formato "mensaje||hash8".
mensaje_cifrado = vigenere_cifrar(mensaje_firmado, clave_simetrica): Llama a la función vigenere_cifrar (también explicada anteriormente) para cifrar el mensaje_firmado (que ya incluye la "firma" hash) utilizando la clave_simetrica generada aleatoriamente. Este es el mensaje que realmente viajará por la red simulada.
Cifrado de la Clave Simétrica con RSA (para el Destino):

- **public_key = G.nodes[destino]['rsa_public']:** Obtiene la clave pública RSA del nodo destino (que fue asignada cuando se creó la red).
- **cipher_rsa = PKCS1_OAEP.new(public_key):** Crea un objeto de cifrado RSA utilizando el esquema de relleno PKCS1_OAEP, que es un método seguro para usar RSA para cifrar datos pequeños (como una clave simétrica).
- **clave_cifrada = cipher_rsa.encrypt(clave_simetrica.encode()):** Cifra la clave_simetrica (primero codificada a bytes) utilizando la clave pública del destino. Solo el nodo destino, que posee la clave privada correspondiente, podrá descifrar esta clave_cifrada.

Impresión de Información de Envío: Las líneas print(...) muestran información sobre el proceso de envío: el origen, destino, el camino, la clave simétrica cifrada (en formato hexadecimal) y el mensaje cifrado Vigenère.




```
# Simula la intercepción por el espía si está activo
if espia is not None:
    print(f"\n⚠️ Nodo espía interceptó el mensaje en nodo {espia} ({G.nodes[espia]['
    print(" - Contenido interceptado:", mensaje_cifrado)
    # Simula que el espía intenta descifrar la clave simétrica (para demostración, a
    try:
        spy_private_key = G.nodes[destino]['rsa_private'] # Espía obtiene la clave p
        spy_clave_descifrada = PKCS1_OAEP.new(spy_private_key).decrypt(clave_cifrada
        print(" - Clave simétrica descifrada (por el espía):", spy_clave_descifrada)
    except Exception as e:
        print(f" - El espía no pudo descifrar la clave simétrica. Error: {e}")

# Obtiene la clave privada del nodo destino
private_key = G.nodes[destino]['rsa_private']
# Descifra la clave simétrica usando la clave privada del destino
clave_descifrada = PKCS1_OAEP.new(private_key).decrypt(clave_cifrada).decode()
# Descifra el mensaje cifrado usando Vigenère con la clave simétrica descifrada
mensaje_descifrado = vigenere_descifrar(mensaje_cifrado, clave_descifrada)
# Verifica la firma digital del mensaje descifrado
valido, mensaje_final = verificar_firma(mensaje_descifrado)

# Imprime información sobre el mensaje recibido
print("\n📥 Mensaje recibido:")
print(" - Descifrado:", mensaje_final)
print(" - Integridad:", "✅ Válido" if valido else "❌ Alterado")

# Devuelve el camino recorrido y el nodo espía (si existe)
return camino, espia
```

Simulación de Intercepción por el Espía (si espia no es None):

- **if espia is not None::** Este bloque de código se ejecuta solo si se seleccionó un nodo espía.
- Muestra un mensaje indicando que el espía interceptó el mensaje y el contenido interceptado (mensaje_cifrado).

- **Simulación de Descifrado por el Espía:** La parte dentro del try...except dentro de este bloque *simula* que el espía tiene la clave privada del *destino* (G.nodes[destino]['rsa_private']). **Importante:** En un ataque MITM real, el espía no tendría la clave privada del destino. Esta parte es solo para demostrar qué pasaría si el espía *pudiera* descifrar la clave simétrica (lo cual solo podría hacer si comprometiera la clave privada del destino, que es muy difícil con RSA). Intenta descifrar la clave_cifrada con la clave privada del destino y muestra la clave simétrica si tiene éxito.

Descifrado en el Nodo Destino:

- private_key = G.nodes[destino]['rsa_private']: Obtiene la clave privada RSA del nodo destino.
- clave_descifrada = PKCS1_OAEP.new(private_key).decrypt(clave_cifrada).decode(): Utiliza la clave privada del destino para descifrar la clave_cifrada. Esto recupera la clave_simetrica original. .decode() la convierte de bytes de nuevo a cadena de texto.

- mensaje_descifrado = vigenere_descifrar(mensaje_cifrado, clave_descifrada): Llama a la función vigenere_descifrar para descifrar el mensaje_cifrado utilizando la clave_simetrica recién descifrada. Esto debería recuperar el mensaje_firmado original ("mensaje||hash8").
- valido, mensaje_final = verificar_firma(mensaje_descifrado): Llama a la función verificar_firma para comprobar si la "firma" adjunta al mensaje descifrado es válida. La función devuelve un booleano (valido) y el mensaje_final (el mensaje original sin la firma).

Impresión de Información de Recepción: Muestra el mensaje_final descifrado y si la Integridad es "✅ Válido" (si valido es True) o "❌ Alterado" (si valido es False).
return camino, espia: La función finaliza devolviendo el camino que siguió el mensaje y el nodo espia (que será None si el espía no estuvo activo)

En resumen, esta función simula el viaje de un mensaje desde el origen hasta el destino, aplicando múltiples capas de seguridad (firma simple, cifrado Vigenère para el mensaje, cifrado RSA para la clave de Vigenère) y mostrando el proceso, incluyendo una posible intercepción simulada. Luego, en el destino, simula el proceso inverso para recuperar el mensaje original y verificar su integridad.

Este código es la preparación inicial dentro de la función de visualización para obtener la información necesaria del grafo (G) antes de empezar a dibujar el mapa.
Aquí está el desglose:

```python
# Nueva función para visualizar la red en un mapa interactivo usando Plotly
def mostrar_red_en_mapa_plotly(G, camino, origen, destino, espia=None):
    """
    Visualiza la red de dispositivos en un mapa interactivo usando Plotly.

    Args:
        G (nx.Graph): El grafo que representa la red de dispositivos.
        camino (list): La lista de nodos que forman el camino del mensaje.
        origen (int): El nodo de origen del mensaje.
        destino (int): El nodo de destino del mensaje.
        espia (int, optional): El nodo espía, si existe.
    """
    # Obtiene las posiciones de los nodos
    pos = nx.get_node_attributes(G, 'pos')
    # Crea etiquetas para los nodos con su número y ciudad
    labels = {n: f"{n} ({G.nodes[n]['ciudad']})" for n in G.nodes}
```

def mostrar_red_en_mapa_plotly(G, camino, origen, destino, espia=None):: Define la función mostrar_red_en_mapa_plotly con los argumentos que ya mencionamos: el grafo G, la lista de nodos que forman el camino del mensaje, los nodos de origen y destino, y el nodo espia si existe.
Docstring (""" ... """): Explica el propósito de la función: visualizar la red en un mapa interactivo usando la librería Plotly. Describe cada uno de los argumentos que recibe la función.
pos = nx.get_node_attributes(G, 'pos'):

- **nx.get_node_attributes(G, 'pos'):** Esta función de networkx es muy útil. Va a través de cada nodo en el grafo G y busca el atributo llamado 'pos'.
- Como recordatorio, cuando creamos el grafo con crear_red_dispositivos, le añadimos a cada nodo un atributo 'pos' que contenía las coordenadas geográficas (latitud, longitud) de la ciudad asignada a ese nodo.
- Esta función recopila todos esos atributos 'pos' de todos los nodos y los devuelve en un diccionario. La clave del diccionario es el índice del nodo (0, 1, 2, etc.), y el valor es la tupla de coordenadas (latitud, longitud).
- **pos = ...:** Asigna este diccionario de posiciones a la variable pos. Ahora, pos es un diccionario donde puedes buscar las coordenadas de cualquier nodo por su número (ej: pos[0] te daría las coordenadas del nodo 0).
- **labels = {n: f"{n} ({G.nodes[n]['ciudad']})" for n in G.nodes}**:
    - Esta es una *comprensión de diccionario* en Python, una forma concisa de crear diccionarios.
    - **for n in G.nodes:** Itera sobre cada nodo (n) en el grafo G.
    - **G.nodes[n]['ciudad']:** Para cada nodo n, accede a sus atributos y obtiene el valor del atributo 'ciudad' (el nombre de la ciudad asignada a ese nodo).
    - **f"{n} ({...})":** Crea una cadena formateada para usarla como etiqueta. La cadena será el número del nodo (n), seguido de un espacio y entre paréntesis el nombre de la ciudad del nodo (G.nodes[n]['ciudad']). Por ejemplo, para el nodo 0 en Bogotá, la etiqueta sería "0 (Bogotá)".
    - **{n: ...}:** Crea un par clave-valor para el diccionario donde la clave es el número del nodo (n) y el valor es la cadena de etiqueta que acabamos de crear.
    - **labels = ...:** Asigna el diccionario resultante a la variable labels. Este diccionario labels ahora contiene la etiqueta que se mostrará junto a cada nodo en la visualización, haciendo más fácil identificar qué nodo es cuál y a qué ciudad corresponde.

Visualización de Red en Mapa Interactivo

En resumen, estas líneas iniciales de la función mostrar_red_en_mapa_plotly se encargan de extraer del grafo la información esencial para la visualización: las coordenadas geográficas de cada nodo (pos) y las etiquetas informativas para cada nodo (labels). Con esta información, la función ya está lista para empezar a construir los diferentes elementos visuales del mapa (las aristas, los nodos, el camino, etc.) utilizando Plotly.

Este bloque de código se enfoca en preparar los datos y crear los objetos visuales (llamados "trazos" o *traces* en Plotly) que representarán las conexiones entre los nodos (aristas) y los nodos mismos en el mapa.

Aquí tienes el detalle:

1

```
# Prepara los datos para las aristas del grafo (sin información de distancia)
edge_x = []
edge_y = []
for edge in G.edges():
    x0, y0 = pos[edge[0]]
    x1, y1 = pos[edge[1]]
    edge_x.extend([x0, x1, None]) # Use extend for cleaner code
    edge_y.extend([y0, y1, None]) # Use extend for cleaner code


# Crea el trazo para las aristas
edge_trace = go.Scatter(
    x=edge_x, y=edge_y,
    line=dict(width=0.5, color='#888'),
    hoverinfo='none',  # No muestra información al pasar el ratón
    mode='lines')

# Prepara los datos para los nodos del grafo
node_x = []
node_y = []
node_text = []
node_color = []
node_size = []
for node in G.nodes():
    x, y = pos[node]
    node_x.append(x)
    node_y.append(y)
    node_text.append(labels[node])
```

2

```
# Crea el trazo para el camino
path_trace = go.Scatter(
    x=path_edge_x, y=path_edge_y,
    line=dict(width=2.5, color='green'),
    hoverinfo='none', # No muestra información al pasar el ratón
    mode='lines',
    name='Camino')

# Crear punto animado para el mensaje que se mueve a lo largo del camino
message_point = go.Scatter(
    x=[pos[camino[0]][0]], y=[pos[camino[0]][1]],
    mode='markers',
    marker=dict(size=10, color='darkgreen'),
    name='Mensaje')

# Crea la figura de Plotly con los trazos
fig = go.Figure(data=[edge_trace, node_trace, path_trace, message_point],
                layout=go.Layout(
                   title='Red de Dispositivos y Camino del Mensaje', # Título actualizado
                   titlefont_size=16,
                   showlegend=True,
                   hovermode='closest',
                   margin=dict(b=20,l=5,r=5,t=40),
                   annotations=[
                    ],
                   xaxis=dict(showgrid=False, zeroline=False, showticklabels=False),
                   yaxis=dict(showgrid=False, zeroline=False, showticklabels=False))
                  )
```

5

```python
    # Asigna colores y tamaños especiales a los nodos de origen, destino y espía
    if node == origen:
        node_color.append('orange')
        node_size.append(20)
    elif node == destino:
        node_color.append('red')
        node_size.append(20)
    elif node == espia:
        node_color.append('purple')
        node_size.append(20)
    else:
        node_color.append('lightblue')
        node_size.append(10)
```

```python
# Crear frames para la animación
frames = []
frames_per_edge = 10 # Esto debe coincidir con la velocidad de animación deseada
total_frames = len(camino) * frames_per_edge
for i in range(total_frames):
    edge_index = min(i // frames_per_edge, len(camino) - 2) # Ensure index is within bounds
    start_node = camino[edge_index]
    end_node = camino[edge_index + 1]
    start_pos = pos[start_node]
    end_pos = pos[end_node]
    t = (i % frames_per_edge) / frames_per_edge
    x = start_pos[0] + (end_pos[0] - start_pos[0]) * t
    y = start_pos[1] + (end_pos[1] - start_pos[1]) * t
    frames.append(go.Frame(data=[go.Scatter(x=[x], y=[y], mode='markers', marker=dict(size=10, col

# Asigna los frames a la figura
fig.frames = frames
```

3

6

```python
# Crea el trazo para los nodos
node_trace = go.Scatter(
    x=node_x, y=node_y, mode='markers+text',
    text=node_text,
    textposition="bottom center",
    hoverinfo='text',
    marker=dict(
        showscale=False,
        colorscale='YlGnBu',
        reversescale=True,
        color=node_color,
        size=node_size,
        colorbar=dict(
            thickness=15,
            title='Conexiones de Nodos',
            xanchor='left',
            titleside='right'
        ),
        line_width=2))

# Crear aristas para el camino recorrido por el mensaje (sin información de distancia)
path_edge_x = []
path_edge_y = []
for i in range(len(camino) - 1):
    x0, y0 = pos[camino[i]]
    x1, y1 = pos[camino[i+1]]
    path_edge_x.extend([x0, x1, None]) # Use extend for cleaner code
    path_edge_y.extend([y0, y1, None]) # Use extend for cleaner code
```

```python
# Agregar botón de reproducción y slider para controlar la animación
fig.update_layout(
    updatemenus=[
        dict(
            type="buttons",
            buttons=[dict(label="Reproducir",
                          method="animate",
                          args=[None, {"frame": {"duration": 100, "redraw": True},
                                       "fromcurrent": True, "transition": {"duration": 0}}],
                          args2=[None, {"frame": {"duration": 100, "redraw": True},
                                        "mode": "immediate", "transition": {"duration": 0}}])],
            pad={"r": 10, "t": 87},
            showactive=False,
            x=0.1,
            xanchor="right",
            y=0,
            yanchor="top"
        )
    ],
    sliders=[dict(steps=[dict(args=[[f.name]], # Corrected args for slider steps
                              label=k,
                              method="animate") for k, f in enumerate(fig.frames)],
                  transition=dict(duration=0),
                  x=0.1,
                  xanchor="left",
                  y=0,
                  yanchor="top")]
)
```

7

```python
# Muestra la figura interactiva
fig.show()
```

 # Prepara los datos para las aristas del grafo (sin información de distancia): Este comentario indica que el siguiente código prepara las líneas que conectan los nodos. No se usa información de distancia para dibujarlas, solo las coordenadas de inicio y fin.
edge_x = [] y edge_y = []: Se inicializan dos listas vacías. edge_x almacenará las coordenadas X (longitud) de los puntos que definen las aristas, y edge_y almacenará las coordenadas Y (latitud).
for edge in G.edges():: Itera sobre cada arista (conexión) en el grafo G. Cada edge es una tupla (nodo_inicio, nodo_fin).
x0, y0 = pos[edge[0]]: Obtiene las coordenadas (x0, y0) del nodo de inicio de la arista (edge[0]) buscando su posición en el diccionario pos que creamos antes. x0 es la longitud y y0 es la latitud.
x1, y1 = pos[edge[1]]: De manera similar, obtiene las coordenadas (x1, y1) del nodo de fin de la arista (edge[1]).
edge_x.extend([x0, x1, None]) y **edge_y.extend([y0, y1, None])**:

- **extend([...]):** Agrega los elementos de la lista proporcionada al final de la lista edge_x o edge_y.
- **[x0, x1, None]:** Para cada arista, se agregan la coordenada X del inicio (x0), la coordenada X del fin (x1), y luego None. Plotly interpreta None en una secuencia de coordenadas como una instrucción para "levantar el lápiz" y no dibujar una línea continua entre el punto anterior y el siguiente. Esto es crucial para dibujar líneas separadas para cada arista. Sin el None, Plotly intentaría conectar el final de una arista con el inicio de la siguiente, creando líneas extrañas.
- Lo mismo se hace para las coordenadas Y (edge_y).

edge_trace = go.Scatter(...): Crea un objeto "trazo" de tipo Scatter de Plotly. Este trazo representará todas las aristas de la red.

- **x=edge_x, y=edge_y:** Le proporciona las listas de coordenadas X e Y que acabamos de preparar.
- **line=dict(width=0.5, color='#888'):** Configura el estilo de la línea: un ancho de 0.5 píxeles y un color gris (#888).
- **hoverinfo='none':** Indica que no se debe mostrar ninguna información emergente cuando el usuario pase el ratón sobre las aristas.
- **mode='lines':** Especifica que este trazo debe dibujarse como líneas conectando los puntos proporcionados.

- **# Prepara los datos para los nodos del grafo**: Comentario que indica la preparación de los datos para los puntos que representan los dispositivos.
- **node_x = [], node_y = [], node_text = [], node_color = [], node_size = []**: Se inicializan varias listas vacías para almacenar las coordenadas X (longitud), coordenadas Y (latitud), el texto de la etiqueta, el color y el tamaño de cada nodo, respectivamente.
- **for node in G.nodes():**: Itera sobre cada nodo en el grafo G.
- **x, y = pos[node]**: Obtiene las coordenadas (x, y) del nodo actual del diccionario pos.
- **node_x.append(x)**, **node_y.append(y)**, **node_text.append(labels[node])**: Agrega la coordenada X, la coordenada Y y la etiqueta (del diccionario labels que creamos antes) a las listas correspondientes para el nodo actual.
- **Bloque if/elif/else para color y tamaño del nodo**: Aquí se decide el color y tamaño de cada nodo basándose en si es el nodo de origen, destino, espia, o un nodo regular.
    - Si el node actual es igual al origen, se le asigna color 'orange' y tamaño 20.
    - Si el node actual es igual al destino, se le asigna color 'red' y tamaño 20.
    - Si el node actual es igual al espia (y espia no es None), se le asigna color 'purple' y tamaño 20.
    - De lo contrario (es un nodo regular), se le asigna color 'lightblue' y tamaño 10.
    - El color y tamaño seleccionados se añaden a las listas node_color y node_size, respectivamente.
- **node_trace = go.Scatter(...)**: Crea un objeto "trazo" de tipo Scatter para representar los nodos.
    - **x=node_x, y=node_y**: Le da las coordenadas de los nodos.
    - **mode='markers+text':** Indica que este trazo debe mostrar tanto marcadores (los puntos de los nodos) como texto (las etiquetas).
    - **text=node_text:** Le proporciona la lista de textos para las etiquetas de los nodos.
    - **textposition="bottom center":** Coloca el texto de la etiqueta debajo del centro del marcador del nodo.
    - **hoverinfo='text':** Indica que cuando el usuario pase el ratón sobre un nodo, se muestre el texto de su etiqueta.
    - **marker=dict(...):** Configura el estilo de los marcadores (los puntos).
        - **showscale=False**: No muestra la barra de escala de color (que a veces se usa con colorscale).
        - **colorscale='YlGnBu', reversescale=True**: Configura una escala de color (aunque no se usa activamente para diferenciar valores en este caso, es parte de la configuración del marcador).
        - **color=node_color: Importante:** Aquí se usa la lista node_color que creamos, lo que permite que cada nodo tenga un color diferente según si es origen, destino, espía, etc.
        - **size=node_size: Importante:** Aquí se usa la lista node_size que creamos, permitiendo que los nodos especiales sean más grandes.
        - **colorbar=dict(...):** Configuración para una barra de color (que, aunque presente en la configuración, no es muy relevante ya que no usamos la escala de color para diferenciar valores).
        - **line_width=2:** Ancho de la línea alrededor de cada marcador de nodo.

 # Crear aristas para el camino recorrido por el mensaje (sin información de distancia): Este comentario indica que se prepararán las líneas que visualizan la ruta específica que tomó el mensaje.

path_edge_x = [] y path_edge_y = []: Se inicializan dos listas vacías para almacenar las coordenadas X (longitud) y Y (latitud) de los puntos que definirán las líneas del camino.
for i in range(len(camino) - 1):: Este bucle itera a través de los nodos en la lista camino. len(camino) es el número total de nodos en el camino. range(len(camino) - 1) genera una secuencia de números desde 0 hasta el penúltimo índice del camino. Esto permite procesar pares de nodos consecutivos en el camino (nodo i y nodo i+1) para dibujar la línea entre ellos.
x0, y0 = pos[camino[i]]: Obtiene las coordenadas (x0, y0) del nodo actual en la iteración (camino[i]) desde el diccionario pos.
x1, y1 = pos[camino[i+1]]: Obtiene las coordenadas (x1, y1) del *siguiente* nodo en el camino (camino[i+1]).
path_edge_x.extend([x0, x1, None]) y **path_edge_y.extend([y0, y1, None])**: Al igual que con las aristas generales del grafo, se agregan las coordenadas del nodo de inicio y fin de este segmento del camino, seguidas por None. Esto asegura que cada segmento del camino (la línea entre dos nodos consecutivos en la ruta) se dibuje como una línea separada en Plotly.
 # Crea el trazo para el camino: Comentario que introduce la creación del objeto visual para la ruta.
path_trace = go.Scatter(...): Crea un objeto "trazo" de tipo Scatter para representar el camino del mensaje.

- x=path_edge_x, y=path_edge_y: Le da las coordenadas de los puntos que definen el camino.
- line=dict(width=2.5, color='green'): Configura el estilo de la línea del camino: un ancho mayor (2.5) y color verde para que destaque.
- hoverinfo='none': No muestra información emergente al pasar el ratón sobre la línea del camino.
- mode='lines': Dibuja el trazo como líneas.
- name='Camino': Asigna un nombre a este trazo, que se usará en la leyenda del gráfico.

 # Crear punto animado para el mensaje que se mueve a lo largo del camino: Comentario que indica la creación del punto que se animará.
message_point = go.Scatter(...): Crea otro objeto "trazo" de tipo Scatter para representar el mensaje como un punto.

- x=[pos[camino[0]][0]], y=[pos[camino[0]][1]]: Inicialmente, coloca este punto en las coordenadas del *primer* nodo del camino (camino[0]). Nota que se usan listas [] porque go.Scatter espera listas de coordenadas, aunque aquí solo haya un punto inicial.
- mode='markers': Dibuja este trazo como un marcador (un punto).
- marker=dict(size=10, color='darkgreen'): Configura el estilo del marcador: tamaño 10 y color verde oscuro.
- name='Mensaje': Asigna un nombre a este trazo para la leyenda. Este punto será el que se animará más adelante.

 # Crea la figura de Plotly con los trazos: Comentario que indica la creación de la figura principal.
fig = go.Figure(data=[edge_trace, node_trace, path_trace, message_point], layout=go.Layout(...)):

- go.Figure(...): Crea el objeto Figure principal que contendrá todos los elementos del gráfico.
- data=[edge_trace, node_trace, path_trace, message_point]: Le pasa una lista con todos los trazos que queremos que se muestren en la figura: las aristas generales, los nodos, el camino específico y el punto del mensaje. Cada uno de estos trazos es una capa visual en el gráfico.
- layout=go.Layout(...): Configura el diseño general del gráfico (título, leyenda, márgenes, ejes).
    - title='Red de Dispositivos y Camino del Mensaje': Establece el título del gráfico.
    - titlefont_size=16: Define el tamaño de la fuente del título.
    - showlegend=True: Muestra la leyenda para identificar los diferentes trazos (Camino, Mensaje, etc.).

- hovermode='closest': Configura cómo se muestran las etiquetas emergentes al pasar el ratón (muestra la del punto más cercano).
- margin=dict(...): Configura los márgenes alrededor del gráfico.
- xaxis=dict(...), yaxis=dict(...): Configuran los ejes X e Y. showgrid=False, zeroline=False, showticklabels=False hacen que los ejes no muestren la cuadrícula, la línea cero ni las etiquetas de los ticks, ya que estamos usando coordenadas geográficas en un mapa.

 # Crear frames para la animación: Comentario que indica la preparación de los pasos para la animación.
frames = []: Se inicializa una lista vacía que almacenará los "frames" o "cuadros" de la animación. Cada frame contendrá la posición actualizada del punto del mensaje en un instante específico del tiempo.
frames_per_edge = 10: Define cuántos cuadros de animación se generarán por cada segmento de arista en el camino. Un número más alto hará la animación más fluida pero generará más frames.
total_frames = len(camino) * frames_per_edge: Calcula el número total de frames necesarios para la animación completa. Es el número de segmentos de arista en el camino (len(camino) - 1, aunque en el bucle se usa len(camino) - 1 para iterar los segmentos) multiplicado por el número de frames por segmento.

for i in range(total_frames):: Este es el bucle principal que itera total_frames veces. Cada iteración de este bucle generará un único "cuadro" o "frame" de la animación. total_frames se calculó antes como el número total de pasos pequeños para que el mensaje recorra todo el camino.
edge_index = min(i // frames_per_edge, len(camino) - 2):

- **i // frames_per_edge:** Calcula a qué segmento de arista en el camino corresponde el frame actual (i). Por ejemplo, si frames_per_edge es 10, los frames 0-9 corresponden a la primera arista, los frames 10-19 a la segunda, y así sucesivamente.
- **len(camino) - 2:** Es el índice del penúltimo nodo en el camino. Hay len(camino) - 1 aristas en total. Los índices de las aristas van de 0 a len(camino) - 2.
- **min(..., ...):** Asegura que edge_index nunca exceda el índice de la última arista válida en el camino.

start_node = camino[edge_index]: Obtiene el nodo de inicio del segmento de arista actual.
end_node = camino[edge_index + 1]: Obtiene el nodo de fin del segmento de arista actual.
start_pos = pos[start_node] y **end_pos = pos[end_node]**: Obtiene las coordenadas (latitud, longitud) del nodo de inicio y fin del segmento de arista actual utilizando el diccionario pos.

t = (i % frames_per_edge) / frames_per_edge:

- **i % frames_per_edge:** Calcula la posición relativa del frame actual (i) dentro del segmento de arista actual. Da un valor entre 0 y frames_per_edge - 1.
- **... / frames_per_edge:** Normaliza este valor a un rango entre 0.0 y (casi) 1.0. t representa el progreso del mensaje a lo largo del segmento de arista actual (0.0 es el inicio, 1.0 sería el final).

x = start_pos[0] + (end_pos[0] - start_pos[0]) * t: Calcula la coordenada X (longitud) del punto del mensaje para este frame. Realiza una interpolación lineal entre la longitud de inicio (start_pos[0]) y la longitud de fin (end_pos[0]), usando t para determinar cuánto se ha movido.
y = start_pos[1] + (end_pos[1] - start_pos[1]) * t: Similar al paso anterior, calcula la coordenada Y (latitud) del punto del mensaje para este frame, interpolando entre la latitud de inicio (start_pos[1]) y la latitud de fin (end_pos[1]).
frames.append(go.Frame(...)): Crea un objeto go.Frame y lo añade a la lista frames. Cada go.Frame define cómo debe verse el gráfico en un momento específico de la animación.

- **data=[go.Scatter(x=[x], y=[y], mode='markers', marker=dict(size=10, color='darkgreen'))]:** Esta es la parte clave del frame. Contiene una lista de trazos que deben actualizarse en este frame. Aquí, solo actualiza la posición del trazo message_point. Crea un nuevo trazo go.Scatter con las coordenadas [x] y [y] calculadas para este frame. Los argumentos mode='markers' y marker=dict(...) se repiten aquí para asegurarse de que el punto se dibuje correctamente en cada frame.

- **name=str(i):** Asigna un nombre único a cada frame (simplemente el número de iteración como cadena).

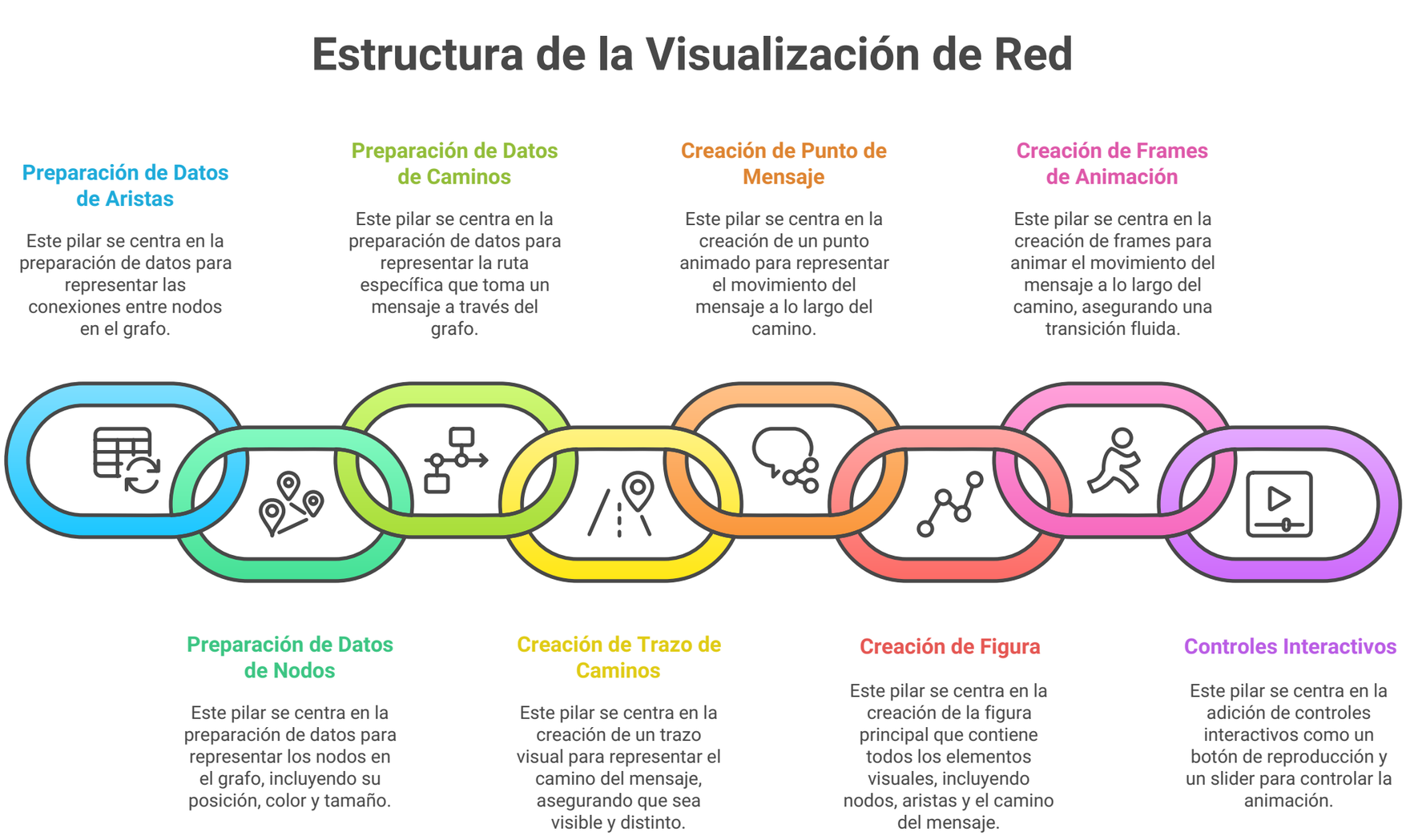
fig.frames = frames: Una vez que el bucle termina y todos los frames se han generado y almacenado en la lista frames, esta línea asigna esa lista a la propiedad frames del objeto fig. Esto vincula la lista de frames a la figura principal, permitiendo que Plotly los use para la animación.
 # Agregar botón de reproducción y slider para controlar la animación: Comentario que indica la configuración de los controles interactivos.
fig.update_layout(...): Este método se utiliza para actualizar el diseño general de la figura, en este caso, añadiendo los controles de animación.

- **updatemenus=[dict(...)]:** Configura la lista de menús de actualización. Aquí se define un solo menú de tipo "buttons" (botones).
  - **type="buttons":** Especifica que es un menú de botones.
  - **buttons=[dict(...)]:** Define la lista de botones dentro del menú. Aquí se define un solo botón.
    - **label="Reproducir":** El texto que se muestra en el botón.
    - **method="animate":** Indica que al hacer clic en este botón, se debe ejecutar el método animate de la figura (iniciar la animación).
    - **args=[None, {"frame": {"duration": 100, "redraw": True}, "fromcurrent":** True, "transition": {"duration": 0}}]: Configura los argumentos para la animación cuando se reproduce desde el principio. Define la duración de cada frame (100 milisegundos), si se redibuja el gráfico en cada frame, etc.
    - **args2=[None, {"frame": {"duration": 100, "redraw": True}, "mode": "immediate", "transition": {"duration": 0}}]:** Argumentos alternativos para el botón (usados al pausar y reanudar).
  - **pad={"r": 10, "t": 87}, showactive=False, x=0.1, xanchor="right", y=0, yanchor="top":** Configuran la posición y apariencia del botón de reproducción.

- **sliders=[dict(...)]:** Configura la lista de sliders. Aquí se define un solo slider.
  - **steps=[dict(...)]:** Define los pasos (marcas) del slider. Cada paso corresponde a un frame de la animación.
    - **dict(args=[[f.name]], label=k, method="animate") for k, f in enumerate(fig.frames):** Esta es una comprensión de lista que crea un diccionario para cada frame en fig.frames.
      - **k, f :** enumerate proporciona tanto el índice (k) como el frame (f).
      - **args=[[f.name]]:** Configura los argumentos para el slider. Al mover el slider a una marca, se llama al método animate de la figura con el nombre (f.name) del frame correspondiente. Esto hace que el gráfico salte directamente a ese frame.
      - **label=k:** Establece la etiqueta de la marca en el slider como el índice del frame.
      - **method="animate":** Indica que mover el slider debe activar el método animate de la figura para ir a ese frame.
  - **transition=dict(duration=0):** Configura la transición cuando se usa el slider (duración 0 significa que salta instantáneamente al frame).
  - **x=0.1, xanchor="left", y=0, yanchor="top":** Configuran la posición y apariencia del slider.

fig.show():

- Una vez que has creado un objeto Figure en Plotly (en tu código, este objeto se llama fig y se crea con go.Figure(...)), este objeto contiene toda la información sobre los datos que quieres graficar (los nodos, las aristas, el camino) y cómo quieres que se vean (colores, tamaños, leyendas, título, diseño).
- El método .show() de este objeto fig le dice a Plotly que renderice esta figura interactiva.
- En entornos como Google Colab o Jupyter Notebooks, fig.show() generará una visualización interactiva directamente en la salida de la celda. Puedes hacer zoom, desplazarte, pasar el ratón sobre los elementos para ver la información (si hoverinfo está configurado correctamente), y en tu caso, interactuar con el botón de reproducción y el *slider* para la animación del mensaje



En resumen, esta parte del código calcula la posición exacta del punto del mensaje en cada pequeño paso del camino, crea un "cuadro" para cada una de esas posiciones, y luego configura los controles (botón de reproducción y slider) que permiten al usuario controlar la animación interactiva en el gráfico de Plotly.

Este fragmento de código corresponde a la función configurar_y_enviar_mensaje_unico**.**
Su propósito es interactuar con el usuario para obtener la información necesaria para enviar un mensaje a través de la red simulada.

Aquí tienes el detalle:

1

2

```python
def configurar_y_enviar_mensaje_unico(G):
    """
    Configura y envía un solo mensaje, solicitando origen, destino y mensaje al usuari
    """
    # Muestra los nodos disponibles y sus ciudades
    print(f"\nNodos disponibles: {[(n, G.nodes[n]['ciudad']) for n in G.nodes]}")

    # --- Entrada para el Mensaje ---
    print("\n--- Configuración del Mensaje ---")
    while True:
        try:
            # Solicita al usuario el nodo origen
            origen = int(input("Nodo origen: "))
            # Verifica si el nodo origen es válido
            if origen not in G.nodes:
                print("Nodo de origen no válido. Por favor, ingresa un nodo de la list
                continue
            # Solicita al usuario el nodo destino
            destino = int(input("Nodo destino: "))
            # Verifica si el nodo destino es válido
            if destino not in G.nodes:
                print("Nodo de destino no válido. Por favor, ingresa un nodo de la lis
                continue
            # Verifica si el nodo origen y destino son diferentes
            if origen == destino:
                 print("El nodo origen y destino no pueden ser el mismo.")
                 continue
            # Sale del bucle si los nodos origen y destino son válidos y diferentes
            break
        except ValueError:
            # Maneja el error si la entrada no es un número entero
            print("Entrada inválida. Por favor, ingresa un número entero.")
        except nx.NetworkXNoPath:
            # Maneja el error si no existe un camino entre los nodos seleccionados
            print(f"No existe un camino entre el nodo {origen} y el nodo {destino}.")
        except Exception as e:
            # Maneja cualquier otro error
            print(f"Ocurrió un error: {e}")
    # Solicita al usuario el mensaje a enviar
    mensaje = input("Mensaje a enviar: ")

    # Solicita al usuario si desea activar el espía
    while True:
        activar_espia_input = input("¿Activar espía (MITM)? (s/n): ").lower()
        # Verifica si la entrada es válida ('s' o 'n')
        if activar_espia_input in ['s', 'n']:
            activar_espia = activar_espia_input == 's' # Asigna True si la entrada es
            break # Sale del bucle después de una entrada válida
        else:
            print("Entrada inválida. Por favor, ingresa 's' o 'n'.")

    # Llama a la función para enviar el mensaje
    camino, espia = enviar_mensaje_unico(G, origen, destino, mensaje, activar_espia)

    # Usar la nueva función de visualización de Plotly
    mostrar_red_en_mapa_plotly(G, camino, origen, destino, espia)
```

def configurar_y_enviar_mensaje_unico(G):: Define la función que toma el grafo G como argumento.
print(f"\nNodos disponibles: {[(n, G.nodes[n]['ciudad']) for n in G.nodes]}"): Imprime una lista de todos los nodos disponibles en la red y la ciudad a la que pertenecen, para que el usuario sepa qué nodos puede elegir.
print("\n--- Configuración del Mensaje ---"): Imprime un encabezado para indicar que comienza la sección de configuración del mensaje.
while True:: Inicia un bucle infinito que se ejecutará hasta que se ingrese información válida para el origen y destino del mensaje. Esto asegura que el programa no continúe hasta que el usuario proporcione nodos válidos.
try...except ValueError...except nx.NetworkXNoPath...except Exception as e:: Este bloque try intenta ejecutar el código de solicitud de entrada. Si ocurre un error (ValueError si la entrada no es un número, nx.NetworkXNoPath si no hay camino entre los nodos, u otra Exception), el bloque except correspondiente captura el error y muestra un mensaje informativo al usuario, permitiéndole intentarlo de nuevo gracias al bucle while True.
Dentro del try (la intención, a pesar de los errores de sintaxis):

- **origen = int(input("Nodo origen: "))**: Pide al usuario que ingrese el número del nodo de origen y lo convierte a un número entero.
- **if origen not in G.nodes:**: Verifica si el número ingresado como origen realmente existe en el grafo. Si no, imprime un mensaje y continue vuelve al inicio del bucle while.
- **destino = int(input("Nodo destino: "))**: Pide al usuario el número del nodo de destino y lo convierte a entero.
- **if destino not in G.nodes:**: Verifica si el nodo destino es válido. Si no, imprime un mensaje y continue vuelve al inicio del bucle.
- **if origen == destino:**: Comprueba que el nodo origen y destino no sean el mismo. Si lo son, imprime un mensaje y continue.
- **break**: Si todas las validaciones anteriores pasan (origen y destino son válidos y diferentes), break sale del bucle while.

mensaje = input("Mensaje a enviar: "): Una vez que origen y destino son válidos, pide al usuario que ingrese el mensaje que desea enviar.
while True: (el inicio de otro bucle, incompleto en tu fragmento): La intención aquí es probablemente pedir al usuario si desea activar la simulación del espía (activar_espia) y validar la entrada ('s' o 'n'). El resto de este bucle y la llamada a las funciones enviar_mensaje_unico y mostrar_red_en_mapa_plotly están en la versión completa de tu código.
activar_espia_input = input("¿Activar espía (MITM)? (s/n): ").lower():

- input("¿Activar espía (MITM)? (s/n): "): Muestra el mensaje "¿Activar espía (MITM)? (s/n): " al usuario y espera a que ingrese algo y presione Enter.
- .lower(): Convierte la entrada del usuario a minúsculas. Esto hace que la comparación posterior sea más flexible, aceptando 's', 'S', 'n' o 'N'.
- activar_espia_input = ...: Almacena la entrada del usuario (en minúsculas) en la variable activar_espia_input.

 # Verifica si la entrada es válida ('s' o 'n'): Un comentario que describe el propósito del siguiente bloque.
if activar_espia_input in ['s', 'n']:: Comprueba si la entrada del usuario (ya en minúsculas) está presente en la lista ['s', 'n']. Es decir, verifica si el usuario ingresó 's' o 'n'.
Dentro del if (si la entrada es válida):

- **activar_espia = activar_espia_input == 's'**: Asigna un valor booleano a la variable activar_espia.
    1. Si activar_espia_input es 's', la comparación 's' == 's' es True, y activar_espia se establece en True.
    2. Si activar_espia_input es 'n', la comparación 'n' == 's' es False, y activar_espia se establece en False.
- **break**: Sale del bucle while True en el que se encuentra este código, porque ya se obtuvo una entrada válida para la activación del espía.

Dentro del else (si la entrada NO es válida):

- **print("Entrada inválida. Por favor, ingresa 's' o 'n'.")**: Imprime un mensaje de error informando al usuario que su entrada no fue correcta. Después de esto, el bucle while True continúa y vuelve a pedir la entrada.

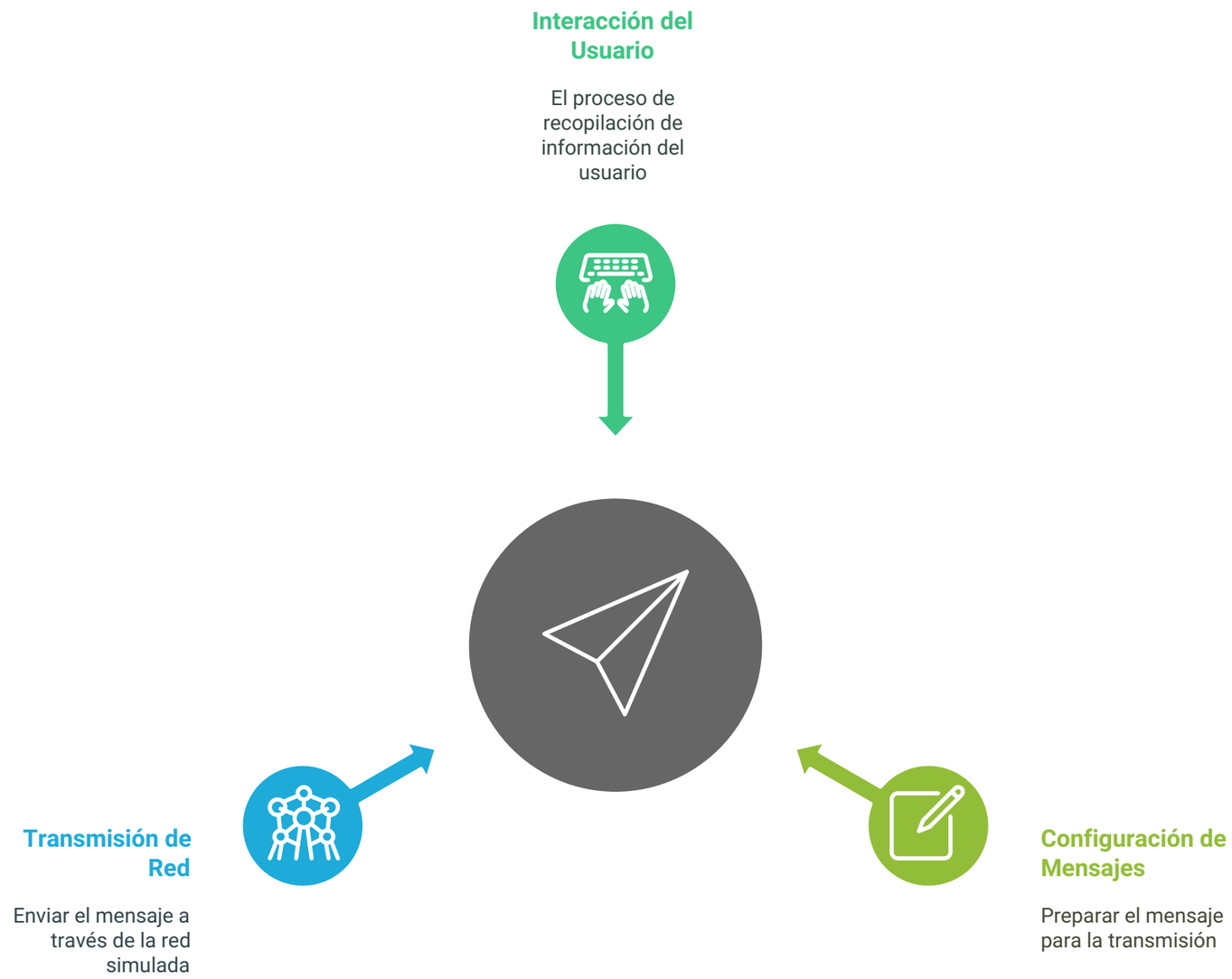
 # Llama a la función para enviar el mensaje: Comentario que indica la siguiente acción.
camino, espia = enviar_mensaje_unico(G, origen, destino, mensaje, activar_espia):

- Llama a la función enviar_mensaje_unico (que explicamos anteriormente).
- Le pasa el grafo G, los nodos origen y destino que el usuario ingresó, el mensaje a enviar, y el valor booleano activar_espia basado en la respuesta del usuario.
- La función enviar_mensaje_unico realiza toda la simulación del cifrado, envío, posible intercepción y descifrado.
- Los valores que devuelve enviar_mensaje_unico (la lista de nodos en el camino y el nodo espía, si hubo uno) se asignan a las variables camino y espia, respectivamente.

 # Usar la nueva función de visualización de Plotly: Comentario que indica la siguiente acción.
mostrar_red_en_mapa_plotly(G, camino, origen, destino, espia):

- Llama a la función mostrar_red_en_mapa_plotly (que también explicamos).
- Le pasa el grafo G, el camino que se calculó, los nodos origen y destino, y el nodo espia (que será None si el espía no estuvo activo).
- Esta función se encarga de crear y mostrar el gráfico interactivo utilizando Plotly, visualizando la red completa, el camino recorrido por el mensaje y, si aplica, el nodo espía.

Proceso de Envío de Mensajes



En resumen, este fragmento de código completa la interacción con el usuario pidiendo la confirmación para activar el espía, valida esa entrada y, finalmente, desencadena las dos acciones principales del programa: la simulación del envío del mensaje (enviar_mensaje_unico) y la visualización interactiva del resultado (mostrar_red_en_mapa_plotly).

Finalmente, llegamos a la explicación de la función main()**. Esta es la función principal de tu script y actúa como el punto de entrada cuando ejecutas el código. Se encarga de la configuración inicial y de presentar el menú interactivo al usuario.**

Aquí tienes el desglose:

1

```python
def main():
    """
    Función principal para ejecutar el simulador de red cifrada con menú interactivo.
    """
    print("\n=== Visualización de Comunicación Segura con Cifrado RSA + Vigenère en un
    # Obtiene el número de ciudades disponibles
    num_ciudades = len(CIUDADES_COORDS)

    G = None # Inicializar G fuera del bucle del menú (el grafo se crea una vez)

    while True:
        # Solo crear el grafo si aún no existe (se crea al inicio de la ejecución)
        if G is None:
            while True:
                try:
                    # Solicita al usuario la cantidad de nodos para la red
                    n = int(input(f"¿Cuántos nodos quieres en la red? (mínimo 3, máxim
                    # Verifica si la cantidad de nodos está dentro del rango permitido
                    if n >= 3 and n <= num_ciudades:
                        G = crear_red_dispositivos(n) # Crea la red de dispositivos
                        break # Sale del bucle si la cantidad de nodos es válida
                    else:
                        print(f"Cantidad de nodos no disponible. Por favor, ingresa un
                except ValueError:
                    # Maneja el error si la entrada no es un número entero
                    print("Entrada inválida. Por favor, ingresa un número entero.")
```

2

```python
        print("\n--- Menú Principal ---")
        print("1. Enviar un mensaje")
        print("3. Salir")
        # Solicita al usuario que seleccione una opción del menú
        opcion_menu_principal = input("Selecciona una opción: ")

        # Procesa la opción seleccionada por el usuario
        if opcion_menu_principal == '1':
            configurar_y_enviar_mensaje_unico(G) # Llama a la función para configurar
        elif opcion_menu_principal == '3':
            print("Saliendo del simulador. ¡Hasta luego!") # Mensaje de despedida
            break # Sale del bucle principal para terminar el programa
        else:
            print("Opción no válida. Por favor, selecciona 1 o 3.") # Mensaje para opc


if __name__ == "__main__":
    main() # Ejecuta la función principal si el script se ejecuta directamente
```

def main():: Define la función principal llamada main.
Docstring (""" ... """): Describe el propósito de la función: ejecutar el simulador de red cifrada con un menú interactivo.
print("\n=== Visualización de Comunicación Segura con Cifrado RSA + Vigenère en una Red Geográfica ==="): Imprime un título llamativo al inicio del programa.
num_ciudades = len(CIUDADES_COORDS): Obtiene el número total de ciudades disponibles en el diccionario CIUDADES_COORDS y lo almacena en num_ciudades. Esto se usa para validar la entrada del usuario sobre cuántos nodos crear.
G = None # Inicializar G fuera del bucle del menú (el grafo se crea una vez): Inicializa la variable G (que representará el grafo o red) a None. Se inicializa fuera del bu bucle principal para asegurar que el grafo se cree solo una vez al inicio de la ejecución del programa, no en cada iteración del menú.
while True:: Inicia el bucle principal del menú. Este bucle se ejecutará indefinidamente hasta que el usuario elija la opción para salir.
if G is None:: Esta condición verifica si el grafo G aún no ha sido creado (lo cual solo ocurrirá la primera vez que se entra en el bucle principal).

- **while True:**: Si G es None, entra en un bucle anidado para solicitar al usuario el número de nodos y crear la red. Este bucle se repite hasta que se ingrese un número de nodos válido.

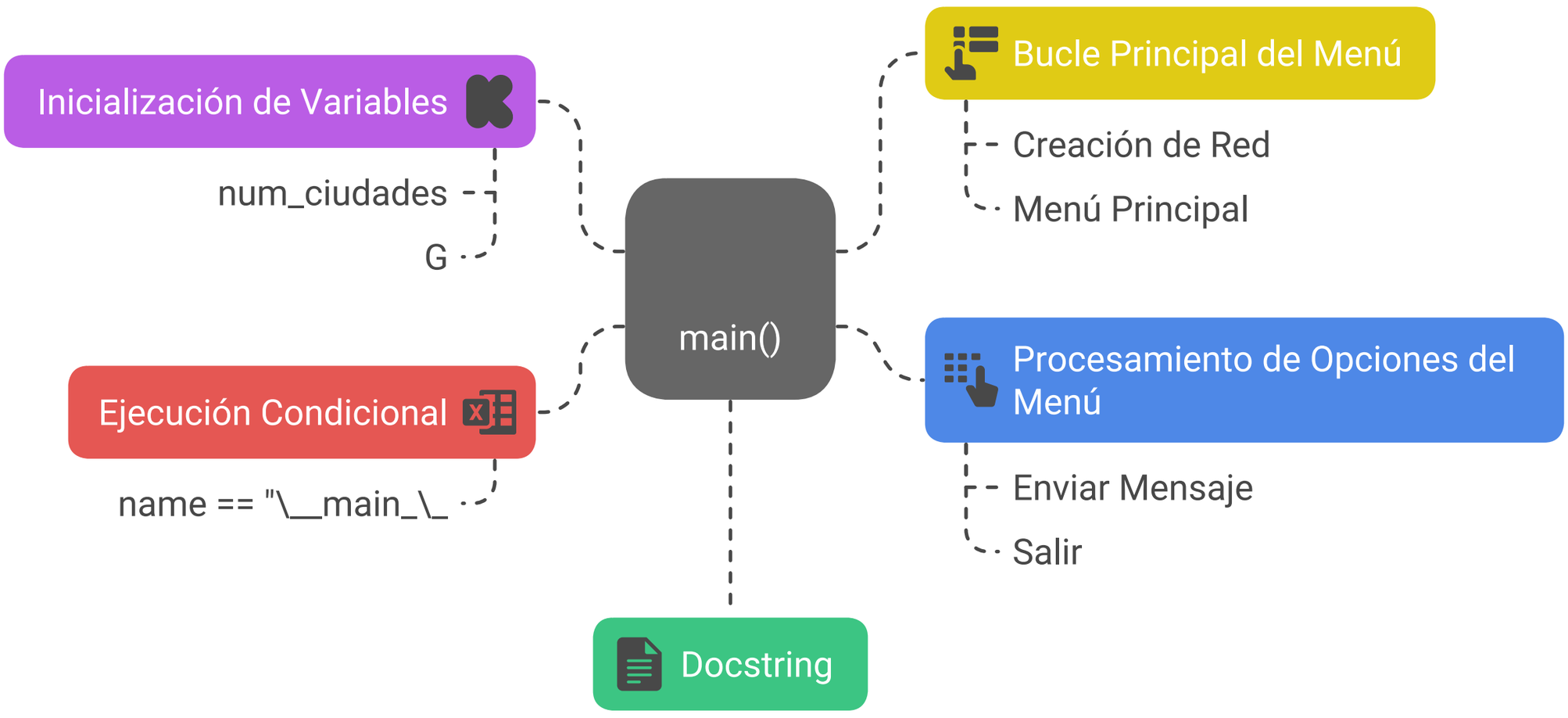
- **try...except ValueError:**: Maneja errores si el usuario no ingresa un número.
- **n = int(input(...))**: Solicita al usuario cuántos nodos quiere en la red, mostrando el rango permitido (mínimo 3, máximo el número de ciudades).
- **if n >= 3 and n <= num_ciudades:**: Valida que el número de nodos ingresado esté dentro del rango permitido.
    - Si es válido: **G = crear_red_dispositivos(n)** llama a la función que crea el grafo y lo asigna a G. **break** sale del bucle anidado de creación del grafo.
    - Si no es válido: **print(...)** muestra un mensaje de error y el bucle anidado continúa.

print("\n--- Menú Principal ---"), **print("1. Enviar un mensaje")**, **print("3. Salir")**: Imprime las opciones del menú principal.
opcion_menu_principal = input("Selecciona una opción: "): Solicita al usuario que ingrese su opción del menú.
Bloque if/elif/else para procesar la opción del menú:

- **if opcion_menu_principal == '1':**: Si el usuario ingresa '1', llama a la función **configurar_y_enviar_mensaje_unico(G)** para iniciar el proceso de envío de un mensaje.
- **elif opcion_menu_principal == '3':**: Si el usuario ingresa '3', imprime un mensaje de despedida y **break** sale del bucle principal while True, terminando la ejecución de main().
- **else:**: Si el usuario ingresa cualquier otra cosa, imprime un mensaje de opción no válida. El bucle principal while True continúa, mostrando el menú de nuevo.

if name == "__main__":: Esta es una construcción estándar en Python. Significa que el código dentro de este bloque solo se ejecutará si el script se está ejecutando directamente (no si se importa como un módulo en otro script).
main(): Si el script se ejecuta directamente, llama a la función main() para iniciar el program

Estructura del Programa de Simulación de Red Cifrada



En resumen, la función main() es el controlador principal de tu script. Inicialmente, pregunta al usuario cuántos nodos desea y crea la red. Luego, presenta un menú simple que permite al usuario elegir entre enviar un mensaje (lo que activa todo el proceso de configuración, envío y visualización) o salir del programa. Continúa mostrando el menú hasta que el usuario elija salir.

Funcionamiento Simulación de Comunicación Segura entre Nodos en una Red Geográfica

```
...
    === Visualización de Comunicación Segura con Cifrado RSA + Vigenère en una Red Geográfica ===
    ¿Cuántos nodos quieres en la red? (mínimo 3, máximo 27): 18

    --- Menú Principal ---
    1. Enviar un mensaje
    3. Salir
    Selecciona una opción: 18
    Opción no válida. Por favor, selecciona 1 o 3.

    --- Menú Principal ---
    1. Enviar un mensaje
    3. Salir
    Selecciona una opción: 1

    Nodos disponibles: [(0, 'Neiva'), (1, 'Cúcuta'), (2, 'Quibdó'), (3, 'Armenia'), (4, 'Cali'), (5, 'Sincelejo')

    --- Configuración del Mensaje ---
    Nodo origen: 7
    Nodo destino: 0
    Mensaje a enviar: colombia
    ¿Activar espía (MITM)? (s/n): n

    📤 Enviando mensaje de nodo 7 a nodo 0:
     - Camino: ['7 (San Andrés)', '0 (Neiva)']
     - Clave cifrada (hex): 03572cb510394aea199e3ef27293708e27d8870e0791cf7be35fb104a520e8d5df0e5a76f4338f6c7dcael
     - Mensaje cifrado: a#y9HWZI3>lBb)DWaj

    📥 Mensaje recibido:
     - Descifrado: colombia
     - Integridad: ✅ Válido
```

Simulación de Comunicación Segura entre Nodos en una Red Geográfica

Esta imagen muestra el flujo completo de una simulación donde se envía un mensaje cifrado desde un nodo de la red hacia otro, utilizando **cifrado híbrido: Vigenère + RSA**.

Detalles de la Simulación:

- **Red configurada con 18 nodos**, cada uno representando una ciudad de Colombia.
- **Menú de opciones**:
    - 1. Enviar un mensaje
    - 3. Salir
- Se eligió:
    - **Origen:** Nodo 7 → **San Andrés**
    - **Destino:** Nodo 0 → **Neiva**
    - **Mensaje:** "Colombia"
    - **Espía (MITM):** No activado (n)

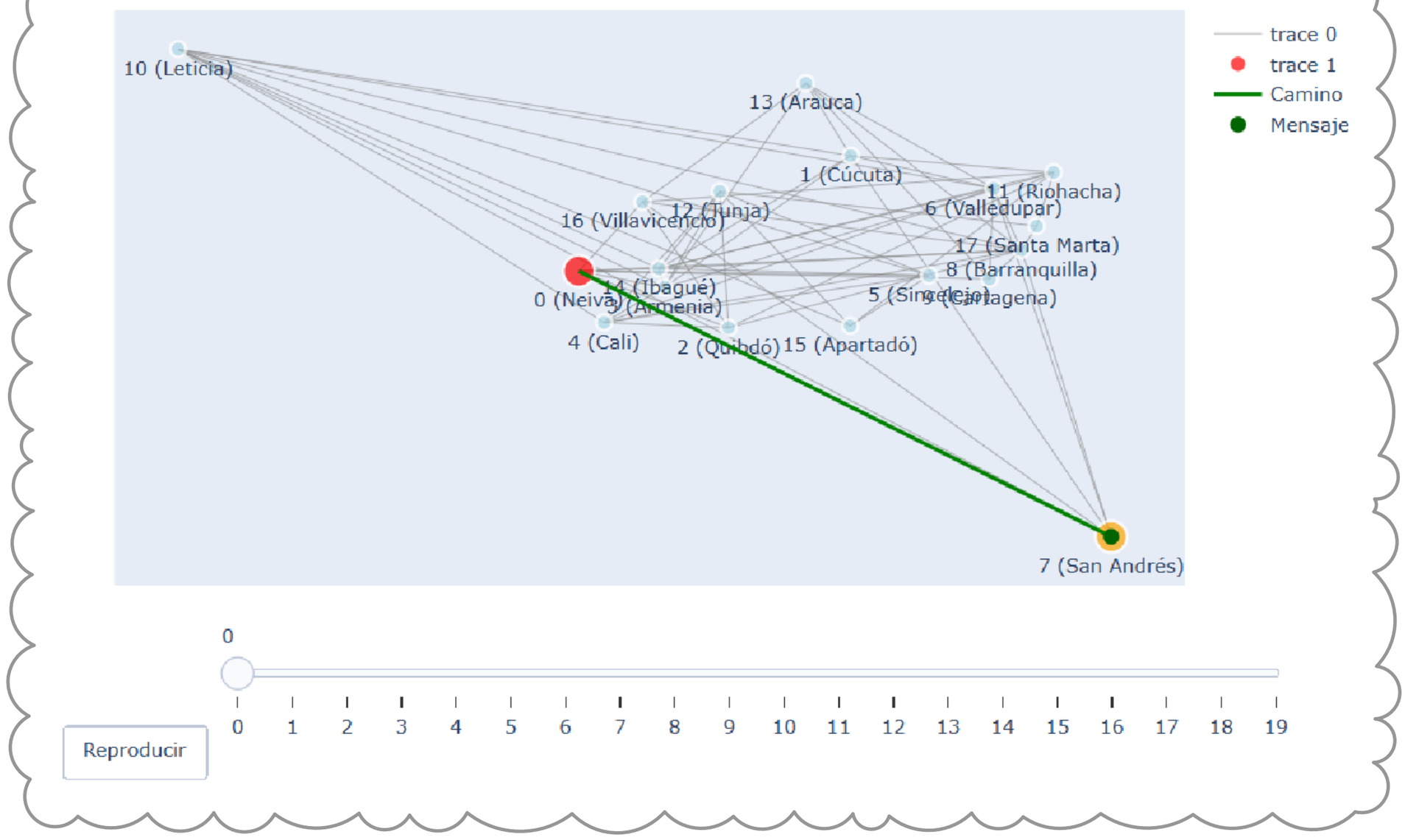
Proceso de Cifrado:

- El mensaje se **firma digitalmente** usando un hash SHA-256 (simulado).
- Luego se cifra con el **algoritmo Vigenère** usando una clave simétrica.
- La clave simétrica se cifra con **RSA (clave pública del nodo destino)**.
- Se muestra:
    - Clave cifrada en hexadecimal.
    - Mensaje cifrado en texto ilegible.

Verificación:

- El nodo de destino (Neiva) **descifra** la clave y luego el mensaje.
- Verifica la firma y confirma:
    - Mensaje descifrado: "Colombia"
    - Integridad del mensaje: **Válido**

Ruta de Comunicación Segura en Red Geográfica



¿Qué representa esta imagen?

Red de ciudades colombianas:

Cada nodo representa una ciudad y está conectado a otras con enlaces simulados (líneas grises). Se trata de una red aleatoria **pero conexa**, es decir, todos los nodos están comunicados.

- Nodos destacados:
  - **San Andrés (Nodo 7):** nodo de **origen** del mensaje.
  - **Neiva (Nodo 0):** nodo **destino**.
  - Punto móvil en el camino: representa el **mensaje en tránsito**, animado sobre el camino.
- Línea verde:

Es el **camino más corto** que sigue el mensaje desde el nodo de origen hasta el destino, calculado automáticamente.

Slider y botón “Reproducir”:

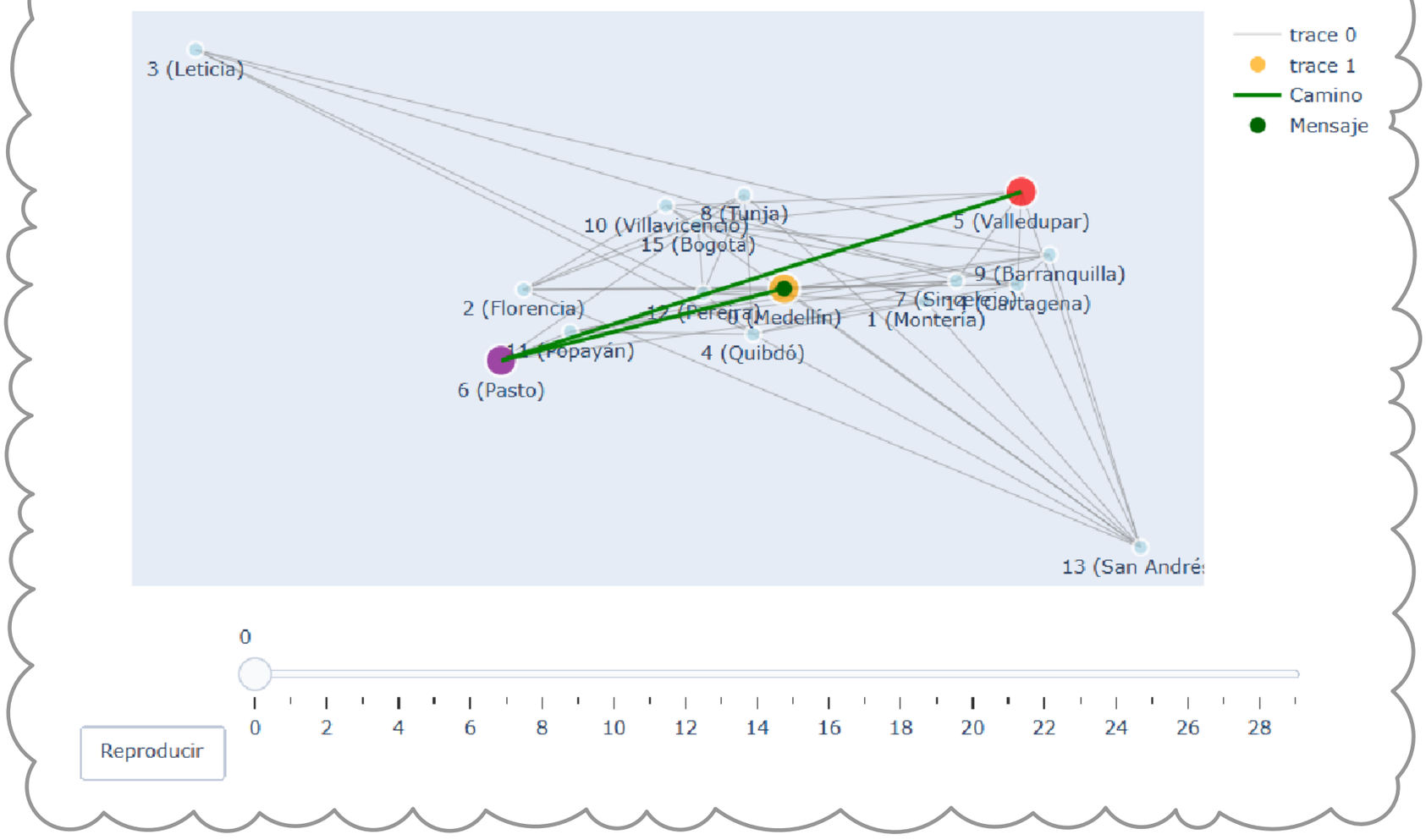
Permite **animar** el recorrido del mensaje paso a paso desde el nodo de origen hasta el destino, como si se tratara de un “viaje seguro” a través de la red ¿Qué hace el sistema?

1. **El mensaje se cifra con Vigenère**, usando una clave aleatoria.
2. Esa clave se cifra con **RSA (clave pública del destino)**.
3. Se **firma digitalmente** para verificar su integridad.
4. Se **envía por la red**, y si se activa el modo espía (MITM), se intercepta.
5. El nodo destino **descifra** el mensaje y verifica la firma.

Conclusión:

Esta imagen representa un sistema visual y educativo para enseñar **criptografía híbrida** (simétrica + asimétrica), redes, seguridad y animación de datos en tiempo real.

“Ruta de Comunicación Segura en Red Geográfica con Espía Simulado (MITM)”



¿Qué se muestra en este gráfico?

Red de nodos:

Cada **nodo** es una ciudad conectada a otras mediante líneas grises (enlaces aleatorios). Esta red es **conexa** y simula una red real de dispositivos.

- Nodo de origen:
  - **Medellín (nodo 12)** está marcado en **naranja**: es donde se origina el mensaje cifrado.
- Nodo de destino:
  - **Valledupar (nodo 5)** está marcado en **rojo**: es donde debe llegar el mensaje y ser descifrado.
- Nodo espía (MITM):
  - **Popayán (nodo 11)** aparece en **morado**: representa un posible atacante "Man-in-the-Middle" que intercepta el mensaje en tránsito.
- Trayectoria del mensaje:
  - La **línea verde gruesa** indica el **camino más corto** seguido por el mensaje cifrado entre origen y destino.
  - El punto (oscuro) es el **mensaje viajando**, y su animación se controla con el botón “Reproducir” y la barra de tiempo inferior.

¿Qué sucede en el sistema?

1. El mensaje es **firmado** digitalmente y cifrado con **Vigenère**.
2. La clave usada es cifrada con **RSA** y enviada con el mensaje.
3. El mensaje viaja por el **camino más corto**.
4. Si hay un espía (MITM activado), este puede ver el mensaje cifrado, pero **no puede descifrarlo** si no tiene la clave privada del destino.
5. El nodo destino recibe el mensaje, lo descifra y **verifica su integridad**.