INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

INGENIERÍA EN SISTEMAS COMPUTACIONALES

SISTEMAS DISTRIBUIDOS

PRÁCTICA 5:

GESTOR DE SERVICIOS Y COMUNICACIONES

GRUPO:
7 CM 1

PROFESOR:
DR. CHADWICK CARRETO ARELLANO

ALUMNOS:

HERNANDEZ MINUTTI BRAULIO
OLIVARES PADILLA JESUS ALEJANDRO

ESCOM
Escuela Superior de Cómputo

Antecedentes

GESTOR DE SERVICIOS Y COMUNICACIONES

Un gestor de servicios y comunicaciones para sistemas distribuidos es una herramienta o software diseñado para administrar y supervisar los servicios y la comunicación entre diferentes componentes o nodos en un entorno distribuido. Proporciona una capa de abstracción que facilita la interacción y el intercambio de información entre los distintos elementos del sistema distribuido.

- ¿Qué es un gestor de servicios y comunicaciones?

Un gestor de servicios y comunicaciones es un componente de software que se encarga de controlar y coordinar los servicios y la comunicación en un entorno distribuido. Actúa como un intermediario entre los diferentes nodos y componentes del sistema, facilitando la interacción y asegurando que los servicios estén disponibles y se comuniquen de manera eficiente.

- ¿Cómo funciona un gestor de servicios y comunicaciones?

El gestor de servicios y comunicaciones tiene varias funciones clave para garantizar el funcionamiento adecuado del sistema distribuido:

1. Descubrimiento de servicios: El gestor permite a los nodos registrarse y descubrir los servicios disponibles en el sistema distribuido. Esto facilita la comunicación entre los componentes y permite la utilización eficiente de los recursos.
2. Coordinación y control: El gestor se encarga de coordinar y controlar la interacción entre los diferentes servicios y nodos. Puede establecer políticas de acceso, prioridades y manejar situaciones de concurrencia para garantizar un funcionamiento ordenado y sin conflictos.
3. Supervisión y monitoreo: El gestor supervisa continuamente el estado de los servicios y los nodos en el sistema distribuido. Puede detectar fallas, rendimiento deficiente o condiciones anómalas, y tomar medidas adecuadas, como reiniciar servicios o redirigir el tráfico, para mantener la estabilidad y disponibilidad del sistema.
4. Gestión de la comunicación: El gestor facilita la comunicación entre los nodos y servicios distribuidos. Puede proporcionar mecanismos de mensajería, enrutamiento eficiente, manejo de colas y asegurar la entrega confiable de mensajes entre los componentes del sistema.

- Ejemplos de utilidad de un gestor de servicios y comunicaciones:

1. En un entorno de computación en la nube, un gestor de servicios y comunicaciones puede ayudar a controlar y coordinar la interacción entre los diferentes servicios y aplicaciones distribuidas, permitiendo una escalabilidad eficiente y un equilibrio de carga adecuado.
2. En sistemas de Internet de las cosas (IoT), un gestor de servicios y comunicaciones puede facilitar la conexión y la comunicación entre los dispositivos distribuidos, permitiendo una integración fluida y un intercambio de datos eficiente.
3. En aplicaciones empresariales distribuidas, un gestor de servicios y comunicaciones puede asegurar la disponibilidad y el rendimiento de los servicios críticos, facilitar la integración de sistemas heterogéneos y optimizar la comunicación entre diferentes sucursales o departamentos.

En resumen, un gestor de servicios y comunicaciones para sistemas distribuidos es una herramienta esencial para administrar y optimizar la interacción entre los componentes distribuidos, asegurando un funcionamiento eficiente y confiable del sistema.

Problemática

Implementación de un gestor de servicios y comunicaciones

Si bien un gestor de servicios y comunicaciones para sistemas distribuidos ofrece numerosos beneficios, su implementación también puede plantear ciertas problemáticas. Algunas de las posibles dificultades incluyen:

Complejidad: La implementación de un gestor de servicios y comunicaciones implica lidiar con la complejidad inherente de los sistemas distribuidos. Coordinar y controlar la comunicación entre nodos y servicios requiere un enfoque meticuloso y una comprensión profunda de la arquitectura y los requisitos del sistema.

Configuración y despliegue: Configurar y desplegar correctamente un gestor de servicios y comunicaciones puede ser un proceso técnico complejo. Requiere definir adecuadamente la topología de red, establecer políticas de acceso, configurar protocolos de comunicación y garantizar la compatibilidad con los diferentes componentes del sistema distribuido.

Escalabilidad: Los sistemas distribuidos a menudo deben manejar un número creciente de servicios y nodos a medida que la demanda aumenta. La implementación de un gestor de servicios y comunicaciones debe abordar los desafíos de escalabilidad, asegurando que el sistema pueda adaptarse y crecer sin afectar negativamente el rendimiento y la disponibilidad.

Tolerancia a fallos: Los sistemas distribuidos son propensos a fallos, ya sea debido a problemas de red, fallas en hardware o software, o incluso ataques malintencionados. Un gestor de servicios y comunicaciones debe incorporar mecanismos robustos de detección y recuperación de fallos para mantener la integridad y la disponibilidad del sistema.

En resumen, la implementación de un gestor de servicios y comunicaciones para sistemas distribuidos puede presentar desafíos relacionados con la complejidad, la configuración, la escalabilidad, la tolerancia a fallos, la seguridad y el mantenimiento. Sin embargo, abordar estas problemáticas adecuadamente puede permitir aprovechar al máximo las ventajas y beneficios que ofrece esta herramienta en entornos distribuidos.

Objetivo

Implementar un gestor de servicios y comunicaciones.

El objetivo de esta práctica es aprender sobre los gestores de servicios y comunicaciones para sistemas distribuidos y comprender sus ventajas en la mejora de la gestión y utilización de los recursos informáticos.

Para lograr este objetivo, se llevarán a cabo las siguientes actividades:

- Estudio de los diferentes gestores de servicios y comunicaciones disponibles. Se explorará su arquitectura, características y funcionalidades.

- Implementación de un entorno de prueba utilizando uno de los gestores de servicios y comunicaciones seleccionados. Se configurarán y desplegarán servicios distribuidos en dicho entorno para comprender los procedimientos y desafíos involucrados en un sistema de cifrado por corrimientos..

En resumen, esta práctica tiene como objetivo aprender sobre los gestores de servicios y comunicaciones para sistemas distribuidos, implementar y comparar diferentes opciones, y comprender sus ventajas en la gestión y utilización de los recursos informáticos en un entorno distribuido.

Desarrollo

Investigación:

El primer paso para el desarrollo de esta práctica fue comprender el funcionamiento de un gestor de este tipo para ello revisamos suficientes fuentes para comprender lo siguiente:

Este gestor tendrá algunas tareas fundamentales, como la gestión de los recursos disponibles para la carga de trabajo, recibir solicitudes de clientes y encolarlas para así mantener un orden dentro de estas, dar respuesta a los clientes.

Estos puntos serán implementados dentro de los 3 tipos de agentes que participan en este proyecto (server, gsc, cliente), cada uno representando una situación real simulada en la que server será encargado de la resolución de trabajos, gsc de lo anteriormente mencionado y el cliente el cual será una petición a los servidores de carga de trabajo.

Implementación:

Server.py

Implementa un servidor XML-RPC que ofrece dos funciones de cifrado y descifrado de texto utilizando el algoritmo de cifrado César. El servidor se ejecuta en la dirección IP "192.168.1.183" y el puerto 3313.

El servidor expone las siguientes funciones:

1. `cifrado(texto, desplazamiento)`: Esta función recibe un texto y un valor de desplazamiento. Aplica el cifrado de corrimientos al texto desplazando cada letra hacia la derecha en el alfabeto en función del valor de desplazamiento. El resultado cifrado se devuelve como una cadena.

2. `descifrado(texto, desplazamiento)`: Esta función recibe un texto cifrado y un valor de desplazamiento. Realiza el descifrado de corrimientos del texto aplicando el desplazamiento inverso utilizado en el cifrado. El texto descifrado se devuelve como una cadena.

3. `linea()`: Esta función simplemente devuelve una cadena indicando que el servidor está en línea en la dirección IP y puerto especificados.

El servidor registra estas funciones y espera consultas de clientes para cifrar o descifrar textos. Una vez que se inicia el servidor, se imprimirá el mensaje "Esperando consultas de cifrado..." en la consola.

El servidor se ejecutará indefinidamente y atenderá las solicitudes de los clientes a través de XML-RPC.

```
import xmlrpc.server

def cifrado(texto,desplazamiento):
    texto = texto.upper()
    alfabeto = "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ"
```

```
    cifrado = ""
    for letra in texto:
        if letra in alfabeto:
            posicion = alfabeto.find(letra)
            nueva_posicion = (posicion + desplazamiento) % 27
            cifrado += alfabeto[nueva_posicion]
        else:
            cifrado += letra
    return cifrado

def descifrado(texto,desplazamiento):
    texto = texto.upper()
    alfabeto = "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ"
    descifrado = ""
    for letra in texto:
        if letra in alfabeto:
            posicion = alfabeto.find(letra)
            nueva_posicion = (posicion - desplazamiento) % 27
            descifrado += alfabeto[nueva_posicion]
        else:
            descifrado += letra
    return descifrado

def linea():
    return "Servidor en linea 192.168.1.75 en puerto 3312"

server = xmlrpc.server.SimpleXMLRPCServer(("192.168.1.183", 3313))
print("Esperando consultas de cifrado...")

server.register_function(cifrado, "cifrado")
server.register_function(descifrado, "descifrado")
server.register_function(linea, "linea")


server.serve_forever()
```

Cliente.py

Implementa un programa que utiliza hilos para realizar la solicitud de cifrado de texto a un servidor remoto a través de XML-RPC. En este se ejecutan las siguientes acciones:

- Se configura el registro de bitácora para almacenar información sobre la ejecución del programa en un archivo llamado "bitacora.log". Se define el nivel de registro como DEBUG para que se muestren todos los mensajes de la bitácora.

```
logging.basicConfig(filename='bitacora.log',
level=logging.DEBUG,filemode='w',
                    format='%(asctime)s | %(levelname)s:%(message)s |
%(threadName)s | %(funcName)s | %(lineno)d|')
```

Se crea un logger y se establece su nivel de registro como DEBUG. También se crea un manejador para mostrar los mensajes de log en la consola y se le aplica un formato específico.

- Se define la función `cifrado(texto, desplazamiento)` que implementa el cifrado del texto dado utilizando un desplazamiento específico.

```
def cifrado(texto,desplazamiento):
    texto = texto.upper()
    alfabeto = "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ"
    cifrado = ""
    for letra in texto:
        if letra in alfabeto:
            posicion = alfabeto.find(letra)
            nueva_posicion = (posicion + desplazamiento) % 27
            cifrado += alfabeto[nueva_posicion]
        else:
            cifrado += letra
    return cifrado
```

- Se define la función `decifrado(cifrado, corrimientos)` que realiza el descifrado de corrimientos del texto cifrado utilizando el número de corrimientos inverso al utilizado en el cifrado.

```
def decifrado(cifrado,corrimientos):
    # código que se ejecutará en el hilo
    cifrado = cifrado.upper()
    alfabeto = "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ"
    decifrado = ""
    for letra in cifrado:
        if letra in alfabeto:
            posicion = alfabeto.find(letra)
            nueva_posicion = (posicion - corrimientos) % 27
            decifrado += alfabeto[nueva_posicion]
        else:
            decifrado += letra
    return decifrado
```

- Se define la función `serverStat()` que se conecta a un servidor remoto a través de XML-RPC y solicita el estado actual de los servidores.

```
def serverStat():
    # Crear proxy
    logging.debug(f"TESTEANDO LOS SERVIDORES ...")
    #print("CONECTANDO CON VM...")
    #http://
    proxy = xmlrpc.client.ServerProxy("http://192.168.1.183:3312/RPC2")
    logging.debug("Conectado para testear")
    libres = proxy.libres()
    print(f"Servidores libres: {libres}")
```

- Se define la función `solicitud(texto, desplazamiento)` que se conecta al servidor remoto a través de XML-RPC y realiza la solicitud de cifrado del texto dado con el desplazamiento especificado.

```
def solicitud(texto,desplazamiento):
    # Crear proxy
    logging.debug(f"Solicitando servidor al gsc desde el cliente ...")
    proxy = xmlrpc.client.ServerProxy("http://192.168.1.183:3312/RPC2")
    logging.debug("Conectado al gsc para solicitar servidor")
    resultado = proxy.solicitud(texto,desplazamiento)
```

```
    return resultado
```

Se muestra el estado actual de los servidores utilizando la función `serverStat()`.

A continuación se presenta el main:

1. Se lee un texto de un archivo llamado "texto.txt" y se solicita al usuario que ingrese el valor de desplazamiento.
2. Se crea un executor de hilos (`ThreadPoolExecutor`) para ejecutar las solicitudes de cifrado en hilos separados.
3. Se inicia un cronómetro para medir el tiempo de ejecución.
4. Se realiza la primera solicitud de cifrado en un hilo utilizando la función `solicitud()`.
5. Se realiza la segunda solicitud de cifrado en otro hilo utilizando la misma función `solicitud()`.
6. Se obtienen los resultados de las solicitudes utilizando el método `result()` de las instancias de Future correspondientes a cada hilo.
7. Se detiene el cronómetro y se calcula el tiempo total de solicitud.
8. Se guarda el texto cifrado en un archivo llamado "cifrado.txt".
9. Se inicia un cronómetro para medir el tiempo de descifrado utilizando únicamente el hilo local.
10. Se realiza el descifrado del texto cifrado utilizando la función `decifrado()`.
11. Se detiene el cronómetro de descifrado y se calcula el tiempo total de ejecución utilizando únicamente el hilo local.
12. Se guarda el texto descifrado en un archivo llamado "descifrado.txt".
13. Se cierra el executor de hilos.
14. Se finaliza la ejecución del programa.

```
logging.debug("Iniciando programa")
logging.debug("Leyendo archivo")
try:
    #Se abre el archivo con codificación utf-8
    with open("texto.txt", "r") as archivo:
        Frase = archivo.read()
except (FileNotFoundError, IOError) as e:
```

```
    logging.error(f"Error al abrir el archivo: {e}")

Desplazamiento = int(input("Ingrese el desplazamiento: "))
print("\n\n")

#Se divide el texto en 2
logging.debug("Dividiendo texto en 2")
mitad = len(Frase)//2
mitad1 = Frase[:mitad]
mitad2 = Frase[mitad:]


#Se ejecutan los hilos
#Crear un executor de hilos
logging.debug("Iniciando executor de hilos")
ejecutaHilo = concurrent.futures.ThreadPoolExecutor()

#Inicia cronómetro de hilo virtual
inicio = time.time()

# Ejecutar la función en un hilo y obtener una instancia de Future

logging.debug("PRIMER CLIENTE")
mi_hilo = ejecutaHilo.submit(solicitud, mitad1, Desplazamiento)

# ##TEST DEL HILO
# resultado = mi_hilo.result()
# print(resultado)


logging.debug("SEGUNDO CLIENTE")
mi_hilo2 = ejecutaHilo.submit(solicitud, mitad2, Desplazamiento)

# ###TEST DEL HILO
# resultado2 = mi_hilo2.result()
# print(resultado2)

#Obtener el resultado de la función

logging.debug("Obteniendo resultado de hilo virtual")
resultado = mi_hilo.result()

logging.debug("Obteniendo resultado de hilo local")
```

```
resultado2= mi_hilo2.result()

fin = time.time()
tiempo_total = fin - inicio
print("Tiempo de solicitud: ", tiempo_total, "segundos")
logging.debug("Fin de executor de hilos")

#Se guarda el texto cifrado
try:
    with open("cifrado.txt", "w") as archivo:
        archivo.write(resultado+resultado2)
    logging.debug("Archivo Cifrados.txt guardado con exito")
except (FileNotFoundError, IOError) as e:
    logging.error(f"Error al abrir el archivo: {e}")



#Inicia cronómetro de descifrado, solo con hilo local
tiempoSolo = time.time()
descif = decifrado(resultado+resultado2, Desplazamiento)
finTiempoSolo = time.time()
tiempoTotalSolo = finTiempoSolo - tiempoSolo
print("Tiempo de ejecución local: ", tiempoTotalSolo, "segundos")


try:
    with open("descifrado.txt", "w") as archivo:
        archivo.write(descif)
    logging.debug("Archivo descifrado.txt guardado con exito")
except (FileNotFoundError, IOError) as e:
    logging.error(f"Error al abrir el archivo: {e}")

#Se cierra el executor de hilos
ejecutaHilo.shutdown()

logging.debug("Fin del programa")
```

En resumen, el código utiliza hilos para realizar solicitudes de cifrado a un servidor remoto y luego guarda los resultados obtenidos. También registra información sobre la ejecución en un archivo de bitácora y muestra mensajes en la consola.

gsc.py

Implementa un servidor GSC (Gestor de Servicios y Comuniciones) que utiliza hilos para monitorear el estado de los servidores remotos y atender las solicitudes de cifrado de texto. El servidor GSC se ejecuta en una dirección IP y puerto específicos y se comunica con los servidores remotos utilizando XML-RPC.

El servidor GSC mantiene una lista de servidores remotos en la variable `listaServidores`, donde cada servidor tiene un indicador de disponibilidad y una dirección URL. Un hilo llamado `Servidores()` se encarga de monitorear continuamente el estado de los servidores remotos. Este hilo intenta establecer una conexión con cada servidor en la lista y actualiza el indicador de disponibilidad en función del resultado de la conexión.

El servidor GSC también implementa la función `solicitud(texto, desplazamiento)` para manejar las solicitudes de cifrado de texto. Esta función busca el primer servidor disponible en `listaServidores` y envía la solicitud a través de XML-RPC. Una vez que se recibe el resultado del cifrado, el servidor GSC marca el servidor como disponible y devuelve el texto cifrado al cliente.

Además, se proporcionan funciones como `serverStat(a)` para probar la conexión con servidores remotos, `ServerFree()` para obtener una lista de servidores disponibles y `handle_connection(client_ip)` para registrar las conexiones de los clientes en un archivo de bitácora.

El programa utiliza un registro de bitácora para almacenar información sobre la ejecución en un archivo llamado "bitacoraGSC.log". También se muestra un mensaje en la consola para indicar que el servidor GSC está en ejecución.

En resumen, el código implementa un servidor GSC que coordina múltiples servidores remotos para atender solicitudes de cifrado de texto. Utiliza hilos para monitorear el estado de los servidores y XML-RPC para la comunicación entre los componentes. El servidor GSC registra eventos importantes en una bitácora y proporciona funciones para interactuar con los clientes.

```
import logging
import xmlrpc.server
import xmlrpc.client
import time
import concurrent.futures


logging.basicConfig(filename='bitacoraGSC.log',
level=logging.DEBUG,filemode='w',
                    format='%(asctime)s | %(levelname)s:%(message)s |
%(threadName)s | %(funcName)s | %(lineno)d|')

listaServidores                                                    =
[[1,"http://192.168.1.183:3313/RPC2"],[1,"http://192.168.1.75:3312/RPC2
"],[1,"http://192.168.1.161:3312/RPC2"]]
listaActivos = []


def Servidores():
    while(1):
        for i in range(len(listaServidores)):
            try:
                serverStat(i)
                listaServidores[i][0] = 1
            except:
                print("Servidor caido")
                       logging.debug(f"Servidor {listaServidores[i][1]}
caido")
                print(listaServidores[i][1])
                listaServidores[i][0] = 0
            if i == 1:
                time.sleep(5)
            #print(listaServidores[i][0])

def serverLocal():
    try:
        logging.debug("Iniciando servidor")
            server = xmlrpc.server.SimpleXMLRPCServer(("192.168.1.183",
3312))
        logging.debug("Servidor iniciado")
    except:
        logging.debug("No se pudo iniciar el servidor")
        print("No se pudo iniciar el servidor")
```

```
    server.register_function(handle_connection, "_dispatch")
    server.register_function(ServerFree, "libres")
    server.register_function(solicitud, "solicitud")
    server.serve_forever()

def solicitud(texto,desplazamiento):
    logging.debug(f"ESTAS EN GSC EN LA FUNCION SOLICITUD ...")
    i=0
    while i<=len(listaServidores):
        logging.debug(f"PROBANDO SERVIDOR {listaServidores[i][1]} ...")
        if (listaServidores[i][0]==1):
               logging.debug(f"EL SERVIDOR {listaServidores[i][1]} ESTA
LIBRE ...")
            listaServidores[i][0]==0
                        logging.debug(f"ENVIANDO SOLICITUD AL SERVIDOR
{listaServidores[i][1]} ...")
            proxy = xmlrpc.client.ServerProxy(listaServidores[i][1])
                                logging.debug(f"CONECTADO AL SERVIDOR:
{listaServidores[i][1]} Y ENVIANDO EL TEXTO...")
            cifrado = proxy.cifrado(texto,desplazamiento)
            #print(resultado)
                 logging.debug(f"RECIBIENDO TEXTO CIFRADO DEL SERVIDOR
{listaServidores[i][1]} ...")
            listaServidores[i][0]==1
               logging.debug(f"EL SERVIDOR {listaServidores[i][1]} ESTA
LIBRE ...")
            i=len(listaServidores)
            logging.debug(f"RETORNANDO TEXTO CIFRADO AL CLIENTE ...")
            return cifrado

        else:
            i=i+1

def serverStat(a):
    # Crear proxy
    #logging.debug(f"Conectando con {listaServidores[a][1]}...")
    #print("CONECTANDO CON VM...")
    #http://192.168.1.75:3312/RPC2
    proxy = xmlrpc.client.ServerProxy(listaServidores[a][1])
    #print("CONECTADO CON VM...\n")
    # Llamar función del servidor
    resultado = proxy.linea()
```

```
    # Imprimir resultado
    #print(resultado)
    #return resultado


#la funcion linea manda a quien la invoca un hola mundo
def ServerFree():
    listaActivos = []
    for i in range(len(listaServidores)):
        if listaServidores[i][0] == 1:
            listaActivos.append(listaServidores[i][1])
    return listaActivos

def handle_connection(client_ip):
    # Registra la conexión en la bitácora
    logging.info(f"Cliente {client_ip} se ha conectado")




print("SERVIDOR GSC CORRIENDO...")


#servers                                                                =
[[1,"http://192.168.1.75:3312/RPC2"],[1,"http://192.168.1.161:3312/RPC2
"]]



logging.debug("Iniciando executor de hilos")
ejecutaHilo = concurrent.futures.ThreadPoolExecutor()

logging.debug("Ejecutando hilo 2")
mi_hilo = ejecutaHilo.submit(Servidores)

logging.debug("Ejecutando hilo 1")
mi_hilo2 = ejecutaHilo.submit(serverLocal)
```

Resultados

Al ejecutar el código anterior tenemos lo siguiente:

```
(cursoml) C:\Users\jalex\Documents\ESCOM IPN\ESCOM 7TH SEM\SISTEMAS
SERVIDOR GSC CORRIENDO...
```

Iniciamos primero el gsc para que este pueda registrar los servidores y su estado

```
Servidor caido
http://192.168.1.183:3313/RPC2
Servidor caido
http://192.168.1.75:3312/RPC2
```

Inicialmente detecta que los servidores se encuentran fuera de línea, pero después se lanzan los servidores y está alerta dejará de emitirse.

```
(cursoml) C:\Users\jalex\Documents\ESCOM IPN\ESCOM 7TH SEM\SISTEMAS DISTRIBUIDOS\P5>python3 server.py
Esperando consultas de cifrado...
192.168.1.183 - - [23/May/2023 17:54:38] "POST /RPC2 HTTP/1.1" 200 -
```

Lanzamos un servidor y nos aparece la leyenda de espera de consultas, después de eso se registran las consultas, en este caso se presenta la de gsc, que pregunta su estado únicamente y se desconecta.

```
(cursoml) C:\Users\jalex\Documents\ESCOM IPN\ESCOM 7TH SEM\SISTEMAS DISTRIBUIDOS\P5>python3 client.py
Servidores libres: ['http://192.168.1.183:3313/RPC2']
Ingrese el desplazamiento:
```

Cuando lanzamos un cliente inmediatamente nos presenta los servidores activos disponibles en el momento de la consulta, después de eso muestra una entrada para el desplazamiento del cifrado.

```
Tiempo de solicitud:  8.225603103637695 segundos
Tiempo de ejecución local:  20.229471445083618 segundos
```

Tiene un tiempo de espera en recibir las respuestas, en este caso podemos notar que cuando se lanza la petición a servidores externos tarda únicamente 8 segundos en cambio en la ejecución local tarda más del doble del tiempo.

Ejemplo de texto claro:
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam ac nunc in elit congue eleifend nec vitae urna. Quisque pellentesque iaculis faucibus. Fusce turpis est, commodo vel libero non, placerat interdum tellus. Nunc semper consequat purus et tempus. In condimentum luctus eros, ac condimentum mauris lobortis id. Etiam placerat magna id tellus efficitur, et consectetur ante dapibus. Pellentesque ullamcorper purus nisl, ac elementum erat ultricies id. Nulla facilisi. Maecenas in massa condimentum mi placerat ultricies ut eu enim. Donec erat mauris, cursus non luctus at, posuere in ligula. Ut mi dui, pretium et diam sit amet, euismod lobortis urna. Proin sagittis ligula eu justo malesuada bibendum.

Ejemplo de texto cifrado:
PTWJQ NUXZQ ITPTW XNY FQJY, HTRXJHYJYZW FINUNXHNRL JPNY. FPNVZFQ FH RZRH NR JPNY HTRLZJ JPJNKJRI RJH ANYFJ ZWRF. VZNXVZJ UJPPJRYJXVZJ NFHZPNX KFZHNGZX. KZXHJ YZWUNX JXY, HTQQTIT AJP PNGJWT RTR, UPFHJWFY NRYJWIZQ YJPPZX. RZRH XJQUJW HTRXJVZFY UZWZX JY YJQUZX. NR HTRINQJRYZQ PZHYZX JWTX, FH HTRINQJRYZQ QFZWNX PTGTWYNX NI. JYNFQ UPFHJWFY QFLRF NI YJPPZX JKKNHNYZW, JY HTRXJHYJYZW FRYJ IFUNGZX. UJPPJRYJXVZJ ZPPFQHTWUJW UZWZX RNXP, FH JPJQJRYZQ JWFY ZPYWNHNJX NI. RZPPF KFHNPNXN. QFJHJRFX NR QFXXF HTRINQJRYZQ QN UPFHJWFY ZPYWNHNJX ZY JZ JRNQ. ITRJH JWFY QFZWNX, HZWXZX RTR PZHYZX FY, UTXZJWJ NR PNLZPF. ZY QN IZN, UWJYNZQ JY INFQ XNY FQJY, JZNXQTI PTGTWYNX ZWRF. UWTNR XFLNYYNX PNLZPF JZ ZXYT QFPJXZFIF GNGJRIZQ.

Ejemplo de texto decifrado:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam ac nunc in elit congue eleifend nec vitae urna. Quisque pellentesque iaculis faucibus. Fusce turpis est, commodo vel libero non, placerat interdum tellus. Nunc semper consequat purus

et tempus. In condimentum luctus eros, ac condimentum mauris lobortis id. Etiam placerat magna id tellus efficitur, et consectetur ante dapibus. Pellentesque ullamcorper purus nisl, ac elementum erat ultricies id. Nulla facilisi. Maecenas in massa condimentum mi placerat ultricies ut eu enim. Donec erat mauris, cursus non luctus at, posuere in ligula. Ut mi dui, pretium et diam sit amet, euismod lobortis urna. Proin sagittis ligula eu justo malesuada bibendum.

Conclusiones

En conclusión, es muy importante que la coordinación de servidores remotos pueda mejorar la escalabilidad y eficiencia de las operaciones, ya que permite distribuir la carga de trabajo entre múltiples servidores. Esto es especialmente útil en aplicaciones que requieren un alto rendimiento y procesamiento paralelo.

El uso de hilos en los códigos permite ejecutar tareas de forma concurrente, lo que mejora la capacidad de respuesta y la eficiencia del sistema. En el primer código, se utilizan hilos para enviar solicitudes de cifrado a diferentes servidores remotos al mismo tiempo, reduciendo el tiempo de procesamiento total. En el segundo código, se emplea un hilo para monitorear el estado de los servidores remotos de forma continua.

La comunicación entre los componentes se realiza mediante el protocolo XML-RPC, que es una forma sencilla y ampliamente utilizada de intercambio de datos entre sistemas distribuidos. Este enfoque permite la integración de diferentes servidores y sistemas, lo que brinda flexibilidad y escalabilidad en el diseño de aplicaciones distribuidas.

El trabajo colaborativo a lo largo de esta práctica mejorar nuestras habilidades en el área de sistemas distribuidos al comprender un componente realmente importante dentro de esta área.