

REDES DE COMPUTADORAS

CURSO 2024

GRUPO 32

Informe - Obligatorio 1

Autores:

EZEQUIEL CAPÓ

SOFÍA CABRERA

LUCAS GONCALVES

Supervisor:

SANTIAGO FREIRE

September 12, 2024

Contents

1	Introducción	2
2	Parte 1	3
2.1	tcpdump	3
2.2	Wireshark	4
2.3	Casos de estudio. Análisis de tráfico de red	5
2.3.1	Analizando paquetes capturados en conexión SSH	5
2.3.2	Captura de tráfico, descarga HTTP	7
2.3.3	Captura de tráfico, descarga HTTPS	8
2.4	Comando Ping	9
2.4.1	Estructura de un mensaje ICMP	10
2.4.2	Caso de estudio, análisis de diagnóstico de red a diferentes sitios .	13
2.4.3	Captura de tráfico, comando ping	14
2.5	Comando Traceroute	15
2.5.1	Descripción de Encabezados	16
2.5.2	Caso de estudio, análisis de diagnóstico de red a diferentes sitios .	18
2.5.3	Replicar funcionalidad de traceroute usando comandos ping	18
3	Parte 2	19
3.1	Funcionamiento	19
3.2	Pruebas	20
4	Parte 3	21
4.1	Análisis de captura de tráfico del cliente	22
4.1.1	Establecimiento de la conexión	22
4.1.2	Invocación y respuesta de procedimientos remotos	23
4.1.3	Envío de notificaciones	25
4.2	Análisis de captura de tráfico de los dos servidores	26
4.2.1	Establecimiento de la conexión	26
4.2.2	Respuesta de procedimientos invocados por el cliente	27
4.2.3	Evaluación de las conexiones	28
5	Conclusión	29
6	Anexos	30
6.1	capturas de casos de pruebas	30

1 Introducción

En el presente trabajo se aborda la implementación de una biblioteca de llamadas a procedimientos remotos (RPC) y una aplicación cliente-servidor, con el objetivo de familiarizarse con conceptos básicos sobre redes, así como con herramientas de diagnóstico y depuración. Posee como finalidad aplicar los conceptos teóricos de las capas de aplicación y transporte, utilizando la API de sockets TCP y la arquitectura de aplicaciones cliente-servidor.

La biblioteca desarrollada permitirá que un servidor publique sus procedimientos y que un cliente los invoque de manera remota, utilizando el protocolo JSON-RPC 2.0 y la biblioteca de sockets(Python) para la transmisión de mensajes a través de TCP. Además, se realizarán pruebas de la biblioteca implementada, utilizando herramientas como tcpdump y Wireshark para capturar y analizar el tráfico de red, y el emulador Mininet para simular una topología de red.

Por ello, el informe se dividirá en 3 partes:

- **Parte 1:** Emplear herramientas de redes como son tcpdump y Wireshark para capturar y analizar tráfico de red.
- **Parte 2:** Desarrollar una biblioteca de llamadas a procedimientos remotos (RPC) usando JSON-RPC 2.0 y sockets TCP.
- **Parte 3:** Ejecutar y evaluar la biblioteca RPC en un entorno emulado con Mininet.

El material de referencia principal durante el desarrollo de todo el informe resulta ser [Redes de computadoras: Un enfoque descendente]

2 Parte 1

2.1 tcpdump

'**tcpdump**' es una herramienta de línea de comandos utilizada para capturar y analizar tráfico de red. [TCPDUMP, 2024] Funciona al interceptar paquetes de datos que atraviesan una interfaz de red en un sistema y permite ver los detalles de esos paquetes en tiempo real o guardarlos en un archivo para análisis posterior.

Algunos comandos son:

1. '**tcpdump -D**': lista todas las interfaces de red disponibles en el sistema que pueden ser utilizadas para capturar tráfico con '**tcpdump**'. Ejemplo:

```
osboxes@osboxes:~$ sudo tcpdump -D
1.enp0s3 [Up, Running]
2.any (Pseudo-device that captures on all interfaces) [Up, Running]
3.lo [Up, Running, Loopback]
4.nflog (Linux netfilter log (NFLOG) interface)
5.nfqueue (Linux netfilter queue (NFQUEUE) interface)
6.usbmon1 (USB bus number 1)
```

- **enp0s3**: Interfaz de red Ethernet (cableada).
 - **any**: Captura tráfico en todas las interfaces.
 - **lo**: Interfaz de loopback para comunicación interna.
 - **nflog**: Interfaz de registro de Netfilter para análisis de tráfico filtrado.
 - **nfqueue**: Interfaz de Netfilter para manipulación de paquetes en cola.
 - **usbmon1**: Monitor de tráfico USB en el bus 1.
2. '**tcpdump -i <interfaz>**': captura todo el tráfico en la interfaz de red especificada. Ejemplo:

```
osboxes@osboxes:~$ sudo tcpdump -i enp0s3
20:53:27.439935 IP 10.0.2.15.54732 > 224.154.80.208.in-addr.arpa.https:
    Flags [P.], seq 1877070825:1877070864, ack 368446436, win 65535, length
    39
```

1. **20:53:27.439935**: La hora en la que se capturó este paquete.
2. **IP 10.0.2.15.54732**: La dirección IP de origen (10.0.2.15) y el puerto de origen (54732).
3. **>**: Indica la dirección del tráfico, de la IP de origen hacia la IP de destino.

4. **224.154.80.208.in-addr.https**: La dirección IP de destino (224.154.80.208) y el servicio de destino (https, interpretado por tcpdump que indica el uso del puerto 443).
 5. **Flags [P.]**: Indica que este paquete tiene activadas las flags PUSH (P) y ACK (.). Esto significa que el remitente está enviando datos con la solicitud de que sean entregados inmediatamente al nivel de aplicación del receptor (PUSH) y está confirmando la recepción de datos anteriores (ACK)
 6. **seq 1877070825:1877070864**: El número de secuencia de TCP para los datos enviados.
 7. **ack¹ 368446436**: El número de confirmación (ACK) de la secuencia recibida.
 8. **win² 65535**: El tamaño de la ventana de recepción, que es 65535 bytes. Este es un valor que el receptor informa al remitente para indicar cuántos bytes está dispuesto a recibir sin enviar otra confirmación.
 9. **length 39**: Longitud del payload³. Indica la carga útil de datos que se está transmitiendo en este paquete. El paquete contiene 39 bytes de datos.
3. **'sudo tcpdump -i <interfaz> -w <archivo.pcap>'**: guarda el tráfico capturado en un archivo '.pcap'. Los archivos PCAP⁴ (Packet Capture) son archivos que contienen datos de red capturados.

2.2 Wireshark

'wireshark' es una herramienta de análisis de red que permite capturar y examinar paquetes de datos en tiempo real.[Wireshark User's Guide, 2024] Ofrece una visualización detallada de cada paquete, soporta una amplia gama de protocolos, y proporciona herramientas avanzadas de filtrado, análisis y estadísticas.

La principal diferencia con `tcpdump` es que dispone de una interfaz gráfica de usuario (GUI) que facilita la navegación, el filtrado y el análisis detallado de paquetes, mientras que la otra se usa mediante línea de comando, lo que puede ser menos intuitivo para algunos usuarios. Es más accesible para el análisis visual y detallado.

¹mecanismo de TCP para confirmar la recepción de datos

²Gestiona el flujo de datos entre el remitente y el receptor, permitiendo que TCP sea eficiente

³Este número coincide con la diferencia entre los números de secuencia 1877070825 y 1877070864

⁴Se puede abrir un archivo .pcap desde wireshark yendo a archivos→abrir y seleccionándolo

2.3 Casos de estudio. Análisis de tráfico de red

En primera instancia capturaremos el tráfico SSH establecido en una conexión remota hacia computadoras de la FING, posteriormente se analizará la descarga de páginas webs HTTP y HTTPS. Se realizarán breves análisis de las capturas buscando responder las interrogantes planteadas en el laboratorio

2.3.1 Analizando paquetes capturados en conexión SSH

Hemos definido que tcpdump realicé una captura de tráfico mientras ingresamos a equipos remotos situados en la Facultad de Ingeniería, dicho acceso vía ssh⁵ quedó almacenado en un archivo .pcap, donde posteriormente lo hemos visualizado mediante la interfaz de Wireshark, arrojó los siguientes resultados

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.2.15	192.168.1.1	DNS	81	Standard query 0x620d A login-ens.fing.edu.uy
2	0.000023	10.0.2.15	192.168.1.1	DNS	81	Standard query response 0x8819 AAAA login-ens.fing.edu.uy
3	0.004318	192.168.1.1	10.0.2.15	DNS	97	Standard query response 0x620d A login-ens.fing.edu.uy A 164.73.44.3
4	0.012446	192.168.1.1	10.0.2.15	DNS	144	Standard query response 0x8819 AAAA login-ens.fing.edu.uy CNAME lulu.fing.edu.uy SOA ns.fing.edu.uy
5	0.012517	10.0.2.15	164.73.44.3	TCP	74	48318 → 22 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM TSval=692007 TSecr=0 WS=128
6	0.024445	164.73.44.3	10.0.2.15	TCP	60	22 → 48318 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
7	0.024464	10.0.2.15	164.73.44.3	TCP	54	48318 → 22 [ACK] Seq=1 Ack=1 Win=29200 Len=0
8	0.024705	10.0.2.15	164.73.44.3	SSHv2	96	Client: Protocol (SSH-2.0-OpenSSH_7.2p2 Ubuntu-4ubuntu2.10)
9	0.024782	164.73.44.3	10.0.2.15	TCP	60	22 → 48318 [ACK] Seq=1 Ack=43 Win=65535 Len=0
10	0.040391	164.73.44.3	10.0.2.15	SSHv2	75	Server: Protocol (SSH-2.0-OpenSSH_8.0)
11	0.040432	10.0.2.15	164.73.44.3	TCP	54	48318 → 22 [ACK] Seq=43 Ack=22 Win=29200 Len=0
12	0.040706	10.0.2.15	164.73.44.3	SSHv2	1390	Client: Key Exchange Init
13	0.040785	164.73.44.3	10.0.2.15	TCP	60	22 → 48318 [ACK] Seq=22 Ack=1379 Win=65535 Len=0
14	0.051816	164.73.44.3	10.0.2.15	SSHv2	1102	Server: Key Exchange Init
15	0.053253	10.0.2.15	164.73.44.3	SSHv2	102	Client: Elliptic Curve Diffie-Hellman Key Exchange Init
16	0.053321	164.73.44.3	10.0.2.15	TCP	60	22 → 48318 [ACK] Seq=1070 Ack=1427 Win=65535 Len=0
17	0.067249	164.73.44.3	10.0.2.15	SSHv2	506	Server: Elliptic Curve Diffie-Hellman Key Exchange Reply, New Keys
18	0.068872	10.0.2.15	164.73.44.3	SSHv2	70	Client: New Keys
19	0.068963	164.73.44.3	10.0.2.15	TCP	60	22 → 48318 [ACK] Seq=1522 Ack=1443 Win=65535 Len=0
20	0.069057	10.0.2.15	164.73.44.3	SSHv2	98	Client:
21	0.069137	164.73.44.3	10.0.2.15	TCP	60	22 → 48318 [ACK] Seq=1522 Ack=1487 Win=65535 Len=0
22	0.079677	164.73.44.3	10.0.2.15	SSHv2	98	Server:
23	0.079711	10.0.2.15	164.73.44.3	SSHv2	130	Client:
24	0.079798	164.73.44.3	10.0.2.15	TCP	60	22 → 48318 [ACK] Seq=1566 Ack=1563 Win=65535 Len=0
25	0.099868	164.73.44.3	10.0.2.15	SSHv2	742	Server:

Figure 1: Captura tráfico SSH

Como es visible en la Figura 1⁶ Los protocolos que interactúan en el proceso son DNS, TCP y SSHv2. De los cuales DNS y SSHv2 pertenecen a la capa de aplicación y TCP a la capa de transporte. Los nodos involucrados en la conexión resultan ser las IPs 10.0.2.15 (cliente) y 164.75.44.3 (servidor). Encontrándose el cliente en nuestra red local, siendo nuestra máquina virtual, y el servidor fuera de esta.

Las primeras 4 líneas emplean el protocolo DNS. La primera línea evidencia una consulta DNS desde 10.0.2.15⁷ (cliente) al servidor DNS en 192.168.1.1 (IP del router de la red de acceso) para resolver la dirección IP asociada al host login-ens.fing.edu.uy. La segunda línea solicita lo mismo, diferenciándose en que la primera consulta 0x620d

⁵para ingresar a los equipos remotos se empleó: `ssh -l <NOMBRE_USUARIO> login-ens.fing.edu.uy`, luego `ssh pcunix66.fing.edu.uy` para acceder a un equipo específico

⁶Los colores en la imagen sirven para discernir entre los tipos de paquetes

⁷Dicha ip correspondiente a la VM

A y la segunda 0x8819 AAAA. En definitiva consulta por la ipv4 (query A) e ipv6 (query AAAA)[DNS AAAA/A] correspondientes al dominio login-ens.fing.edu.uy, donde el hexadecimal es solo un identificador de la consulta.

Las siguientes 2 líneas, la 3era y la 4ta, corresponden a la respuesta brindada por parte del router (192.168.1.1). En la 3era línea nos retorna la ipv4 y en la 4ta nos retorna un CNAME (alias canónico)

Posteriormente, las líneas 5, 6 y 7 establecen el "Handshake de TCP". En la línea 5, el cliente envía un paquete con el flag [SYN] para iniciar la sincronización. En la línea 6, el servidor responde con un paquete que contiene los flags [SYN, ACK], indicando que acepta la conexión. Finalmente, en la línea 7, el cliente envía la confirmación con el flag [ACK], completando el proceso y estableciendo la conexión TCP.

Luego, el conjunto de paquetes enviados desde la línea 8 a 25 describe el establecimiento de una conexión SSH segura y pasaje de datos. Se intercambia información de protocolo (líneas 8-11). Se negocian y establecen las claves mediante la definición de Elliptic Curve Diffie-Hellman⁸. (líneas 12-17). En la línea 18 el cliente solicita la clave y en la línea 19 el servidor confirma este proceso. Las líneas restantes (20-25) establecen el intercambio de datos cifrados⁹.

En este proceso, se intercalan los protocolos SSH y TCP, como se observa en Figura 1 TCP se utiliza para el transporte confiable de los paquetes, mientras que SSH maneja la seguridad, cifrando la comunicación.

De esta manera, por más que se intercepten los paquetes, no se podrá descifrar su contenido al estar encriptados. Seguramente las líneas 20-25 contengan la contraseña ingresada para acceder a SSH con el usuario correspondiente

Dado esto, al ejecutar cualquier comando como son 'ls' o 'top', teniendo la sesión SSH activa, no se podrá distinguir ningún tipo de dato enviado, es el cometido del protocolo SSH. La siguiente Figura 2 corresponde a la ejecución de dichos comandos, donde no se evidencia ninguna distinción. En definitiva, se pueden conseguir los datos de los paquetes fácilmente, pero estos están encriptados con una complejidad computacional muy elevada, en la práctica resulta imposible descifrarlos

⁸No es el algoritmo de cifrado, si no, un protocolo de intercambio de claves, en la siguiente web se explica el funcionamiento superficial <https://protecciondatos-lopd.com/empresas/algoritmo-diffie-hellman/>,

⁹El algoritmo de cifrado posiblemente sea SHA256, por los datos desglosados de los paquetes

No.	Time	Source	Destination	Protocol	Length	Info
51	6.024299	10.0.2.15	164.73.44.3	SSHv2	202	Client:
52	6.024448	164.73.44.3	10.0.2.15	TCP	60	22 → 48318 [ACK] Seq=2254 Ack=1711 Win=65535 Len=0
53	6.335944	164.73.44.3	10.0.2.15	SSHv2	82	Server:
54	6.335963	10.0.2.15	164.73.44.3	TCP	54	48318 → 22 [ACK] Seq=1711 Ack=2282 Win=34584 Len=0
55	6.336060	10.0.2.15	164.73.44.3	SSHv2	166	Client:
56	6.336156	164.73.44.3	10.0.2.15	TCP	60	22 → 48318 [ACK] Seq=2282 Ack=1823 Win=65535 Len=0
57	6.508640	164.73.44.3	10.0.2.15	SSHv2	682	Server:
58	6.545226	10.0.2.15	164.73.44.3	TCP	54	48318 → 22 [ACK] Seq=1823 Ack=2910 Win=36680 Len=0
59	6.558805	164.73.44.3	10.0.2.15	SSHv2	98	Server:
60	6.558815	10.0.2.15	164.73.44.3	TCP	54	48318 → 22 [ACK] Seq=1823 Ack=2954 Win=36680 Len=0
61	6.558914	10.0.2.15	164.73.44.3	SSHv2	498	Client:
62	6.559059	164.73.44.3	10.0.2.15	TCP	60	22 → 48318 [ACK] Seq=2954 Ack=2267 Win=65535 Len=0
63	6.572885	164.73.44.3	10.0.2.15	SSHv2	326	Server:
64	6.609283	10.0.2.15	164.73.44.3	TCP	54	48318 → 22 [ACK] Seq=2267 Ack=3226 Win=38776 Len=0
65	6.626666	164.73.44.3	10.0.2.15	SSHv2	162	Server:
66	6.626676	10.0.2.15	164.73.44.3	TCP	54	48318 → 22 [ACK] Seq=2267 Ack=3334 Win=38776 Len=0
67	12.286547	10.0.2.15	164.73.44.3	SSHv2	98	Client:
68	12.286726	164.73.44.3	10.0.2.15	TCP	60	22 → 48318 [ACK] Seq=3334 Ack=2311 Win=65535 Len=0
69	12.299580	164.73.44.3	10.0.2.15	SSHv2	98	Server:
70	12.299588	10.0.2.15	164.73.44.3	TCP	54	48318 → 22 [ACK] Seq=2311 Ack=3378 Win=38776 Len=0
71	12.677917	10.0.2.15	164.73.44.3	SSHv2	98	Client:
72	12.678108	164.73.44.3	10.0.2.15	TCP	60	22 → 48318 [ACK] Seq=3378 Ack=2355 Win=65535 Len=0
73	12.689263	164.73.44.3	10.0.2.15	SSHv2	122	Server:

Figure 2: Captura tráfico SSH

2.3.2 Captura de tráfico, descarga HTTP

Se ha capturado el tráfico al descargar la página <http://www.columbia.edu/~fdc/sample.html>, se capturaron varios paquetes correspondientes a dicha acción, donde se involucran protocolos TCP (capa transporte), DNS (capa aplicación) y HTTP (capa aplicación). Para minimizar la contaminación de paquetes se aplicó un filtro ¹⁰. La siguiente Figura 3 evidencia que el protocolo HTTP empleado es 1.1 y que se obtuvieron los archivos: sample.html, picture-of-something.jpg y favicon.ico. Al seleccionar los paquetes es posible visualizar su contenido, en el caso del HTML nos brinda todo su código, en la imagen e icono no nos brinda una visualización directa.

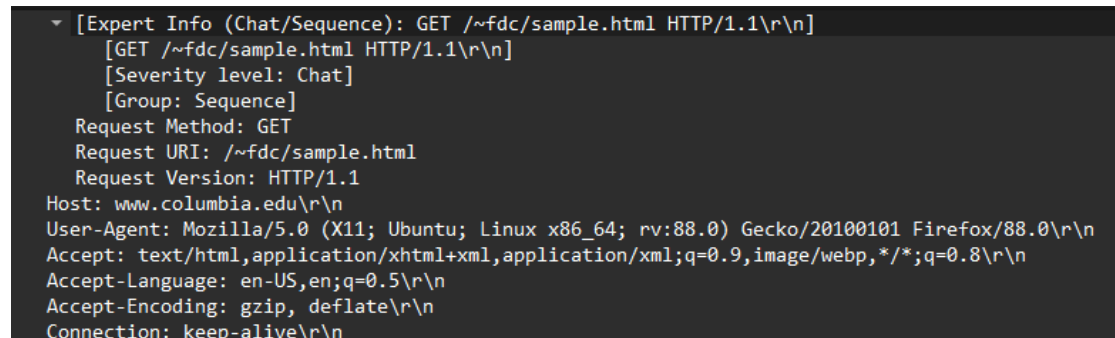
Se puede diferenciar que los pedidos del cliente (10.0.2.15), resultan en métodos GET, pide los componentes de la página, y las respuestas del servidor (162.159.128.59, ubicado fuera de la red local), resultan en los datos pedidos exitosamente, ya que retorna el código "200 OK".

No.	Time	Source	Destination	Protocol	Length	Info
9	0.022478	10.0.2.15	162.159.128.65	HTTP	405	GET /~fdc/sample.html HTTP/1.1
17	0.273186	162.159.128.65	10.0.2.15	HTTP	5531	HTTP/1.1 200 OK (text/html)
41	0.352725	10.0.2.15	162.159.128.65	HTTP	671	GET /~fdc/picture-of-something.jpg HTTP/1.1
45	0.471347	10.0.2.15	162.159.128.65	HTTP	653	GET /favicon.ico HTTP/1.1
53	0.651741	162.159.128.65	10.0.2.15	HTTP	60	HTTP/1.1 200 OK (image/x-icon)
77	0.718370	162.159.128.65	10.0.2.15	HTTP	1136	HTTP/1.1 200 OK (JPEG JFIF image)

Figure 3: Captura tráfico HTTP

¹⁰el filtro ingresado es "http", es decir, solo obtiene paquetes con ese protocolo, es posible filtrar por puerto tcp.port == 80, pero también se obtienen paquetes con TCP

Ya que el protocolo HTTP no dispone de ningún tipo de encriptación, es posible capturar todos los datos transportados. Por otra parte, la solicitud del cliente nos brinda más datos (Figura 4) los lenguajes (accept-languages), archivos (accept) y encoding (accept-encoding) ¹¹ aceptados, y el User-Agent (aplicación desde la que se realiza la solicitud)¹²



```

[Expert Info (Chat/Sequence): GET /~fdc/sample.html HTTP/1.1\r\n]
[GET /~fdc/sample.html HTTP/1.1\r\n]
[Severity level: Chat]
[Group: Sequence]
Request Method: GET
Request URI: /~fdc/sample.html
Request Version: HTTP/1.1
Host: www.columbia.edu\r\n
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:88.0) Gecko/20100101 Firefox/88.0\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8\r\n
Accept-Language: en-US,en;q=0.5\r\n
Accept-Encoding: gzip, deflate\r\n
Connection: keep-alive\r\n

```

Figure 4: Información de paquete HTTP capturado, línea nro 9

Otros datos obtenidos son los media-type transferidos resultan en: text/html, image/x-icon, image/JPEG.

Y así como se visualizó los encodings aceptados por parte del cliente que realiza la solicitud, el servidor también dispone de estos datos, en este caso el encoding del servidor resulta en gzip, lo cual coincide con uno de los aceptados por el cliente. Por lo cual no habrían problemas de codificación

2.3.3 Captura de tráfico, descarga HTTPS

Ahora capturamos el tráfico de descarga de la página <https://www.fing.edu.uy/>. En este caso, los datos capturados viajan cifrados, y vemos que el protocolo evidenciado ya no resulta en HTTP, si no, en OCSP¹³, y dicho protocolo OCSP pertenece a la capa aplicación, otros protocolos involucrados son el DNS, TCP y TLS(Transport security layer, pertenece a la capa de transporte) como se puede visualizar en Figura 5, se muestra que el cliente es la IP 10.0.2.15 y que se comunica con dos servidores en Internet, resultan en las ips 200.40.28.46 y 142.250.79.67, indicando que los componentes se almacenan en diferentes hosts.

¹¹algoritmos de compresión sobre los media-type. (ejemplos: gzip, br y deflate)
[MDN Web Docs, 2024]

¹²\r : retorno de carro y \n: salto de línea, convenciones de HTTP

¹³el filtro empleado en Wireshark sigue siendo http y OCSP(Online Certificate Status Protocol) es el protocolo que proporciona información sobre el estado de revocación de certificados digitales

En lo que respecta a comparar la descarga con HTTP, los datos de accept-encoding, accept-language, user-agent y la versión de HTTP(1.1) son las mismas. Pero el método de la solicitud de parte del cliente en vez de GET como lo era en HTTP, resulta en POST y ahora el resultado "200 OK" no está más retornado en el campo info, se debe acceder a la información desglosada del paquete para visualizarlo. Evidentemente, al estar cifrado no podemos visualizar el código HTML de la página ni sus componentes, los mismos aparecen como "DATA FILE" y caracteres hexadecimales.

Por otro lado, el content-encoding del servidor no se visualiza por ningún lado y los media type tampoco son visibles. Lo único que sabemos es que el cliente realiza dos solicitudes y el servidor las responde de forma satisfactoria. Análogo a lo que sucede en el caso de HTTP sin cifrar, donde se realizan dos GET, uno para el HTML y otro para la imagen, solo que en este caso no hay información, acerca de cuál es la data que solicita el cliente, al capturar el paquete.

No.	Time	Source	Destination	Protocol	Length	Info
23	0.116794	10.0.2.15	200.40.28.46	OCSP	433	Request
25	0.125842	200.40.28.46	10.0.2.15	OCSP	943	Response
444	0.569352	10.0.2.15	142.250.79.67	OCSP	430	Request
446	0.711033	142.250.79.67	10.0.2.15	OCSP	756	Response

Figure 5: Captura tráfico HTTPS, datos

Finalizando esta sección, puede surgirnos la duda, de como se obtiene el final de la secuencia de los paquetes capturados, Wireshark tiene varias maneras de evidenciar esto, una manera es ingresando a los datos de la línea, donde el encabezado puede tener campos como content-length y siguiendo el flujo (nros de secuencia), sumando los length(longitudes/largos) de las trazas del flujo. Es posible identificar el fin del stream(fluido); otra manera es ingresando a los datos del paquete y visualizando si contiene campos [NEXT SEQUENCE] que hagan referencia a las consecuentes líneas que prosiguen la comunicación, otra manera es identificando FLAGS, como [FIN] que indica que la conexión se cierra, incluso en Wireshark mediante los filtros disponibles es posible seguir una secuencia de protocolos y examinar su traza, dando click derecho → seguir y seleccionando el tipo de protocolo que se desea recorrer, luego examinando los paquetes individualmente se puede evidenciar su flujo.

2.4 Comando Ping

'ping' es una herramienta de línea de comandos que se puede encontrar en cualquier sistema operativo con conectividad de red, sirve como una prueba para ver si se puede

acceder a un dispositivo a través de la red. El comando ping envía varias solicitudes de eco, generalmente cuatro o cinco, y presenta al usuario el resultado de cada una de ellas, explicitando si la solicitud recibió una respuesta exitosa, cuantos bytes se recibieron en respuesta, el tiempo de vida (TTL) y el tiempo que se tardó en recibir la respuesta, junto con otras estadísticas sobre la pérdida de paquetes y los tiempos de ida y vuelta.

La herramienta ping utiliza la solicitud de eco y los mensajes de respuesta que genera dentro del protocolo de mensajes de control de internet (**ICMP**). Este es un protocolo en la capa de red que utilizan los dispositivos de red para diagnosticar problemas de comunicación en la red. Se utiliza principalmente para determinar si los datos llegan o no a su destino a su debido tiempo. Este protocolo es parte del conjunto de protocolos IP (Internet Protocol).

2.4.1 Estructura de un mensaje ICMP

Encabezado IP Todo el tráfico que pasa por Internet tiene encabezados IP. Un encabezado IP es cómo un paquete conoce dónde viajar a través de Internet. Muestran información como la versión IP, la longitud del paquete, la fuente y el destino.

0-3	4-7	8-15	16-18	19-31
Versión	Tamaño Cabecera	Tipo de Servicio	Longitud Total	
Identificador			Flags	Posición de Fragmento
Tiempo de vida		Protocolo	Suma de Control de Cabecera	
Dirección IP de Origen				
Dirección IP de Destino				
Opciones				Relleno

Figure 6: Formato del encabezado IP

- **Version** (4 bits): Puede variar entre (0100) o (0110) dependiendo si se utiliza IP versión 4 (IPv4) o IP versión 6 (IPv6). Este campo describe el formato de la cabecera utilizada.
- **Tamaño del encabezado** (4 bits): Longitud del encabezado.
- **Tipo de servicio** (8 bits): Indica una serie de parámetros sobre la calidad de

servicio deseada durante el tránsito por una red.

Bits 0 a 2: Prioridad. Bit 3: Retardo. 0 = normal ; 1 = bajo. Bit 4: Rendimiento. 0 = normal; 1 = alto. Bit 5: Fiabilidad. 0 = normal; 1 = alta. Bit 6-7: No usados. Reservados para uso futuro.

- **Longitud total** (16 bits): El tamaño total, en octetos, del paquete de datos, incluyendo el tamaño del encabezado y el de los datos.
- **Identificador** (16 bits): Identificador único del paquete de datos.
- **Flags** (3 bits): Especifica valores relativos a la fragmentación de paquetes.
- **Posición de fragmento** (13 bits): En paquetes fragmentados indica la posición que ocupa el paquete actual dentro del datagrama original.
- **Tiempo de vida** (8 bits): Indica el máximo número de enrutadores que un paquete puede atravesar.
- **Protocolo** (8 bits): Indica el protocolo de las capas superiores al que debe entregarse el paquete.
- **Header Checksum** (16 bits): Se recalcula cada vez que algún nodo cambia alguno de sus campos.
- **Dirección IP de origen** (32 bits)
- **Dirección IP de destino** (32 bits)
- **Opciones** (Variable): campo no obligatorio, que los nodos deben saber interpretar.
- **Relleno** (Variable): Utilizado para asegurar que el tamaño, en bits, de la cabecera es un múltiplo de 32. El valor usado es el 0.

Encabezado ICMP

El ICMP inicia después del encabezado IPv4. Todos los paquetes ICMP tendrán un encabezado de 8 bytes y la sección de datos de tamaño variable.

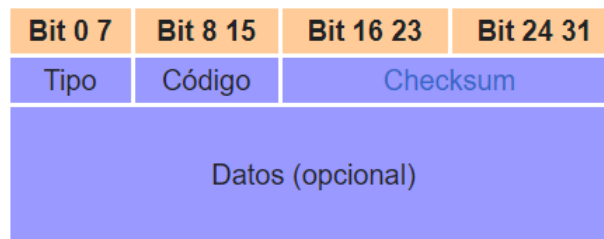


Figure 7: Formato del encabezado IMPC

- **Tipo:** Tipo de ICMP.
- **Código:** Subtipo al tipo dado.
- **Checksum:** Datos de comprobación de errores.

Solicitud de eco (Echo Request)

Es un mensaje de control que se envía a un host con la expectativa de recibir de él una respuesta de eco (Echo Reply).

Byte 0	Byte 1	Byte 2	Byte 3
Type (8 = IPv4, ICMP; 128 = IPv6, ICMP6)	Código	Header Checksum	
Identificador		Número de secuencia	
Datos			

Figure 8: Formato de la solicitud de eco

Respuesta de eco (Echo Reply)

Es un mensaje generado como respuesta a un mensaje de solicitud de eco.

Byte 0	Byte 1	Byte 2	Byte 3
Type (0 = IPv4, ICMP; 129 = IPv6, ICMP6)	Código	Header Checksum	
Identificador		Número de secuencia	
Datos			

Figure 9: Formato de la respuesta de eco

Los datos incluidos en el Echo Request deben estar siempre en los datos del Echo Reply.

2.4.2 Caso de estudio, análisis de diagnóstico de red a diferentes sitios

Al ejecutar el comando ‘ping’, en la salida se mostrarán las siguientes estadísticas:

- Dirección IP del host de destino.
- Número de secuencia ICMP de cada paquete enviado.
- Tiempo de ida y vuelta de cada paquete.
- Tiempo de vida de cada paquete Al pasar por cada router este valor se decrementa en uno.
- Paquetes enviados, recibidos y perdidos.
- Tiempo total de la interacción.
- Tiempo mínimo, promedio y máximo de ida y vuelta.
- Desviación estándar, medida de la variabilidad del RTT (Round-Trip Time) a lo largo del tiempo.

```
ping -c 5 www.antel.com.uy
```

Al ejecutar el comando ping hacia la red de servidores de Antel se pudo observar que las solicitudes de eco enviadas no tuvieron una respuesta correspondiente con la información requerida dado que, cómo se nos indica en consola, la totalidad de las solicitudes se perdieron y no llegaron a destino. Esto sucede porque estos servidores web están configurados para no responder a solicitudes ICMP por razones de seguridad. Existen amenazas cómo ataques DoS o inundación de protocolo de control de mensajes de Internet (ICMP) que consisten en sobrecargar el ancho de banda de un router de red o una dirección IP objetivo, o sobrecargar la capacidad de un dispositivo para reenviar tráfico al siguiente salto de procesamiento posterior al saturarlo con paquetes ICMP elaborados. A medida que el dispositivo intenta responder, todos sus recursos (memoria, capacidad de procesamiento, velocidad de interfaz) se consumen y ya no puede atender solicitudes o usuarios legítimos.

```
ping -c 5 www.google.com
```

La dirección IP del host es 142.251.134.4, el tiempo de vida de cada paquete enviado es de 114 y el RTT promedio es de 19.003ms. Tiempos de ida y vuelta bajos son deseables dado que indican baja latencia y mejor rendimiento de la conexión de red. No hubo pérdida de paquetes y el tiempo total de la interacción fue de 4007ms. En general, se observa que la conexión de red con los servidores de Google es rápida y eficiente dado que tuvimos un bajo índice de tiempos de respuesta al enviar los paquetes.

```
ping -c 5 registro.br
```

La dirección IP del host es 200.160.2.3, el tiempo de vida de los paquetes enviados es de 245 y el RTT promedio es de 76.177ms, siendo el máximo de 132.530ms. Esto indica que existe cierto incremento en el tiempo de ida y vuelta de los paquetes, lo que puede ser causado por la lejanía física entre los servidores involucrados dado que el sitio web indicado es de origen brasileño. Existen otros factores que pueden influir en el tiempo de respuesta de un servidor, tales como sobrecarga de trabajo sobre el mismo o infraestructura de red de baja calidad.

```
ping -c 5 zadna.org.za
```

La dirección IP del host es 129.232.249.120, el tiempo de vida de cada paquete enviado es de 46 y el RTT promedio es de 358.983ms, siendo el mínimo de 332.475ms. Se observa un notorio incremento en el tiempo de ida y vuelta de los paquetes, probablemente causado por la distancia existente entre los servidores involucrados, pues el host de destino está ubicado en Sudáfrica. Los paquetes enviados a estos destinos viajan por enlaces transoceánicos y esto produce una demora en los RTT que se ve reflejada al hacer diagnósticos de red como el que estamos haciendo.

2.4.3 Captura de tráfico, comando ping

Utilizando Wireshark, analizamos la captura de tráfico de red al ser ejecutado el comando ping tomando como ejemplo el sitio web de Google.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.1.7	142.251.134.36	ICMP	98	Echo (ping) request id=0x03ef, seq=1/256, ttl=63 (reply in 2)
2	0.055050	142.251.134.36	192.168.1.7	ICMP	98	Echo (ping) reply id=0x03ef, seq=1/256, ttl=115 (request in 1)
3	1.000923	192.168.1.7	142.251.134.36	ICMP	98	Echo (ping) request id=0x03ef, seq=2/512, ttl=63 (reply in 4)
4	1.065640	142.251.134.36	192.168.1.7	ICMP	98	Echo (ping) reply id=0x03ef, seq=2/512, ttl=115 (request in 3)
5	2.001037	192.168.1.7	142.251.134.36	ICMP	98	Echo (ping) request id=0x03ef, seq=3/768, ttl=63 (reply in 6)
6	2.020616	142.251.134.36	192.168.1.7	ICMP	98	Echo (ping) reply id=0x03ef, seq=3/768, ttl=115 (request in 5)
7	3.002819	192.168.1.7	142.251.134.36	ICMP	98	Echo (ping) request id=0x03ef, seq=4/1024, ttl=63 (reply in 8)
8	3.100794	142.251.134.36	192.168.1.7	ICMP	98	Echo (ping) reply id=0x03ef, seq=4/1024, ttl=115 (request in 7)
9	4.004912	192.168.1.7	142.251.134.36	ICMP	98	Echo (ping) request id=0x03ef, seq=5/1280, ttl=63 (reply in 10)
10	4.089633	142.251.134.36	192.168.1.7	ICMP	98	Echo (ping) reply id=0x03ef, seq=5/1280, ttl=115 (request in 9)

Figure 10: Captura de tráfico con ping

Para simplificar la tarea de análisis y visualizar mejor los mensajes enviados y recibidos, se filtró el tráfico capturado en la red por el protocolo ICMP, como ya vimos es el utilizado en el comando ping. Enviamos 5 solicitudes de eco (Echo Request), de los cuáles tuvimos 5 respuestas de eco (Echo Reply). Se observa que la dirección IP del host de destino tiene una pequeña variación en la dirección IP, esto se debe a que el mensaje fue recibido por un host distinto en la misma red. También podemos observar la presencia del protocolo ICMP en cada mensaje transmitido entre los servidores. Además de eso figuran los tiempos de vida (TTL) para cada paquete, el número de secuencia (**seq**) y el identificador (ID) hexadecimal que corresponde a la sesión de ping (este valor ayuda a emparejar las solicitudes con sus respuestas).

2.5 Comando Traceroute

'**traceroute**' es una herramienta de red utilizada para determinar la ruta, terminal a terminal, que toma un paquete desde un dispositivo a otro. El usuario especifica el nombre del host de destino, el programa del host de origen envía al destino varios paquetes especiales. A medida que estos paquetes se dirigen a su destino, pasan a través de una serie de routers. Cuando un router recibe uno de estos paquetes especiales, devuelve al origen un mensaje corto que contiene el nombre y la dirección del router. Este comando hace uso de 2 protocolos:

1. **ICMP(Internet Control Message Protocol)**: este protocolo es utilizado para enviar mensajes de control y diagnóstico en la capa de red. Los mensajes ICMP son usado por dispositivos como routers para reportar errores o condiciones inusuales que ocurren en la entrega de paquetes.
2. **UDP(User Datagram Protocol)**: es un protocolo de transporte, el mismo no está orientado a la conexión, por lo que no tiene lugar en procedimientos de negociación antes de que los procesos comiencen a comunicarse. Cuando un proceso envía un mensaje a un socket UDP, este protocolo no ofrece ninguna garantía de que el

- **ID Number:** número arbitrario utilizado por el originador del Paquete de Salida para identificar los mensajes ICMP Traceroute. NO está relacionado con el número de ID en el encabezado IP.
- **Dirección IP del Originador:** dirección IP del originador del Paquete de Salida. Campo necesario para que los enrutadores sepan dónde enviar el mensaje ICMP Traceroute para los paquetes de Retorno, así como para los paquetes de salida que tienen una opción de ruta de origen.

Formato del mensaje ICMP Traceroute.

0	8	16	24
Type		Code	Checksum
ID Number		unused	
Outbound Hop Count		Return Hop Count	
Output Link Speed			
Output Link MTU			

Figure 12: Formato encabezado Traceroute ICMP

- **Tipo:** 30
- **Código:** 0 indica que el paquete de salida fue reenviado con éxito. 1 significa que no hay ruta para el paquete de salida; paquete descartado.
- **Checksum:** Es la suma complementaria de 16 bits de las palabras de 16 bits en el encabezado. Para calcularlo, el campo debe estar previamente configurado a cero.
- **ID Number:** El número de identificación copiado de la opción IP Traceroute del paquete que causó el envío de este mensaje Traceroute. No está relacionado con el número de identificación en el encabezado IP.

2.5.2 Caso de estudio, análisis de diagnóstico de red a diferentes sitios

```
tracert www.antel.com.uy14
```

En el resultado obtenido observamos que el IP del host es 200.40.1.182. También vemos que a partir del salto 8 las solicitudes de tiempo de espera se agotan antes de alcanzar el destino final, el motivo de este suceso es el mismo que el de por qué no se obtiene respuesta al ejecutar el comando ping hacia el mismo dominio.

```
tracert www.google.com
```

El IP del host es 142.250.79.100. Luego del salto 5 se observan algunos "Request timed out", lo que nos sugiere que existen problemas de conectividad o que algunos routers no responden a solicitudes ICMP. Se alcanza el destino final en el salto 11 con tiempos de respuesta de 13-21ms.

```
tracert registro.br
```

Observamos que el IP del host es 200.160.2.3. Existen algunos saltos donde se agota el tiempo de espera de la solicitud, lo que puede ser debido a un firewall o un router que no responde a la solicitud ICMP. El destino final se alcanza en el salto 12, con tiempos de respuesta de 29-99ms.

```
tracert zadna.org.za
```

El IP del host es 129.232.249.120. También se presentan saltos donde se agotó el tiempo de espera, los motivos pueden ser los mismos que los del comando anterior. En el salto 14 se alcanza el destino final con tiempos de respuesta de 387-411ms.

2.5.3 Replicar funcionalidad de traceroute usando comandos ping

La forma de replicar el funcionamiento de traceroute utilizando únicamente el comando ping es ajustando el tiempo de vida (TTL). Comenzamos inicializando TTL = 1, por ejemplo: `ping -t 1 www.google.com`. De esta forma se envía el paquete al primer router. Luego se va incrementando gradualmente el valor de TTL para descubrir más

¹⁴El resultado de todas las ejecuciones traceroute serán en realidad el resultado del comando tracert, ya que al ejecutar el comando en WSL2 o en la Virtual Box los paquetes no llegaban a destino.

saltos, hasta alcanzar el destino final del mismo.

De esta forma se emula el comportamiento de `traceroute` al ver cada router intermedio en la ruta que toma el paquete hasta el destino final.

3 Parte 2

Para esta parte, se definió y ejecutó el código en python, correspondiente a una aplicación cliente-servidor empleando la biblioteca de sockets¹⁵ [Sockets Python, 2024], donde el servidor brinda métodos alojados en él mismo, y el cliente solicita esos métodos sin saber como los consigue. Dichos Sockets se encuentran ubicados entre la capa aplicación y la capa de transporte (En el modelo TCP/IP).

Se emplea JSON RPC 2.0 como protocolo de envío de datos.[JSON RPC 2.0, 2013]. A su vez, se creó un set de pruebas, las cuales identifican todos los errores especificados en la transferencia JSON RPC.¹⁶, estos test contemplan también casos de éxito.

3.1 Funcionamiento

El cliente establece la conexión con los servidores, mediante un STUB, para ello se importa la biblioteca `jsonrpc_redes`, la cual brinda las instancias necesarias para delegar la conexión y acceder a los métodos solicitados alojados en el servidor, mediante la definición de sockets cliente, `socket.connect`, luego parsea los datos que se desean enviar, nombre del procedimiento y parámetros. Adicionalmente, se puede definir una bandera `notify = True/False`, que indica si se desea enviar una notificación, la misma está especificada en el documento [JSON RPC 2.0, 2013]. En definitiva, las `notify = True` no obtienen respuestas del servidor, son simples mensajes. Luego de enviar los datos parseados en JSON con la especificación RPC2, envía los datos, `socket.sendall`, y espera la respuesta, `socket.recv`.

Por otro lado, el servidor, obtiene la conexión, definiendo la IP y el puerto que enlaza, `socket.bind`, y escuchando solicitudes, `socket.listen`, si obtiene datos, los acepta, `socket.accept`, recibe la request (`recv`) y la parsea, para dejarla en formato legible internamente, procesando todo los datos, distinguiendo los casos posibles (errores, casos de éxito).

Se definieron dos hilos de ejecución correspondientes a los dos servidores instanciados, los hilos corresponden a la ejecución del método `serve()`, que es el que escucha las solicitudes y las procesa. Y también se definió concurrencia al ejecutar los métodos del

¹⁵Puede profundizar en como funcionan los sockets en <https://www.rfc-editor.org/rfc/rfc147>

¹⁶ej: error -32600, mensaje: invalid request

servidor correspondientes a la solicitud del cliente. Las conexiones resultan no persistentes, se decidió embeber en la lógica de la biblioteca el control de close y connects dinámicos, es decir, siempre al ejecutar un método cierra la conexión del socket, luego detecta si un socket está abierto o no, y lo abre en consecuencia, las llamadas de los métodos son estáticas, es decir ya tienen definidas acceder a un servidor, siendo conn1 la conexión al servidor1 y conn2 al servidor2¹⁷ ¹⁸. Para las pruebas los servers fueron alojados en el localhost con 2 puertos diferentes, (esto para las locales, fuera de la red de mininet, que si tuvimos que definir las ips como se solicitan), cada servidor posee un set de métodos distintos.

Se definieron 5 archivos .py, un clientTest.py que levanta 3 conexiones con servidores distintos (dos servidores que implementamos nosotros y uno externo) y que configura todos los casos de prueba con un cubrimiento total de la especificación jsonrpc, luego un archivo serverOne.py y otro serverTwo.py, que levantan los servidores con sus métodos correspondientes, un archivo serverBase.py que establece el template de nuestras implementaciones de servidores, es decir, define servidores por herencia de un ServerBase, que tiene las configuraciones necesarias para el funcionamiento y conexión de sockets, y toda la lógica esta implementada en la carpeta jsonrpc_redes, donde dentro hay un archivo jsonrpc_redes.py y ahí se establecen las clases necesarias que componen todo el esqueleto y funcionamiento de la implementación. **jsonrpc_redes.py** que mencionamos en el párrafo anterior

3.2 Pruebas

Se definió un set de pruebas exitosas con llamadas a todos los procedimientos disponibles en los servidores, y otras llamadas que abordan los casos de error con mensajes especificados según [JSON RPC 2.0, 2013]. En todo momento se respeta el formato establecido, incluyendo las definiciones de los identificadores "error: code: , message:" y "response:" los cuales no pueden ir juntos.

Se adjunta una visualización de la ejecución de las pruebas realizadas

```
osboxes@osboxes:~/redes2024_ob1$ python3 .\CompartidaVM\Obligatorio1\client.py
Iniciando pruebas de casos sin errores.
=====
EXIT0: Pruebas de casos sin errores completadas.
=====
Iniciando pruebas de casos con errores.
Lanzo excepcion, metodo sin definir: {'code': -32601, 'message': 'Procedimiento
no encontrado'}
```

¹⁷connS1 es la conexión con el servidor ajeno. (implementación de la biblioteca jsonrpc que no nos pertenece)

¹⁸Se añadió una interrupción mediante una excepción para cortar la ejecución del servidor

```
Lanzo excepcion, Error de parseo : {code: -32700, message: Error de parseo}
Lanzo excepcion, Parametros invalidos: {'code': -32602, 'message': 'Parametros
invalidos'}
Lanzo excepcion, Solicitud Invalida: {'code': -32600, 'message': 'Solicitud
Invalida '}
Lanzo excepcion, Error Interno: {'code': -32603, 'message': 'Error Interno'}
=====
EXIT0: Pruebas de casos con errores completadas.
=====
EXIT0: Todas las pruebas completadas.
```

4 Parte 3

En esta sección pondremos a disposición nuestra implementación de la biblioteca **jsonrpc_redes** en un emulador de redes (Mininet), donde ya tenía una configuración previa (Routers, servidor, cliente, topología).

Desde un cliente se desea ejecutar procedimientos que no se encuentran implementados localmente, sino que están disponibles en un servidor remoto.

Las pruebas no solo rigen para nuestra implementación, también se incluye la conexión a un servidor ajeno con métodos definidos, los servidores son expuestos en las ips dadas en la topología siguiente, donde se diferencian mediante los puertos que exponen para postular los métodos, el archivo serverOne.py levanta dos servidores, diferenciando las postulaciones de los métodos por puertos distintos.

Luego de realizadas nuestros casos de prueba, añadimos los casos de prueba extra ajenos, esto para contemplar que tan complementaria es nuestra implementación y si convive correctamente en un ambiente modular, también se aceptaron múltiples conexiones clientes, las mismas realizaban solicitudes a sus métodos de manera concurrente, todos los casos de prueba han culminado con éxito y son debidamente expuestos en Anexos .

A continuación explicaremos el funcionamiento del intercambio de información entre los componentes de la red mediante el análisis de tráfico.

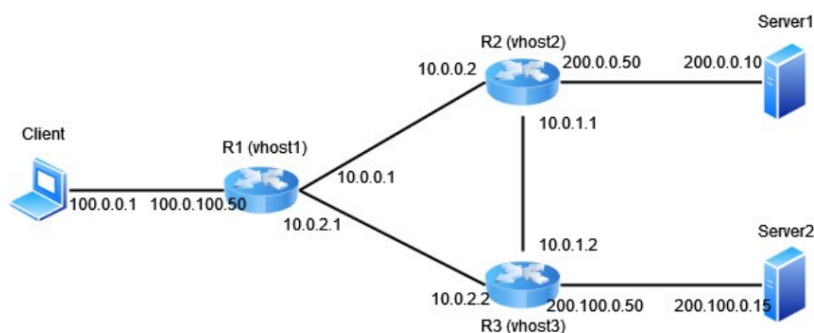


Figure 13: Topología de prueba

19

4.1 Análisis de captura de tráfico del cliente

4.1.1 Establecimiento de la conexión

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	100.0.0.1	200.0.0.10	TCP	74	33348 → 8087 [SYN] Seq=0 Win=29200 L
2	0.365259	b2:13:ca:24:29:19	Broadcast	ARP	42	Who has 100.0.0.1? Tell 100.0.0.50
3	0.365328	8e:8c:af:6e:be:d0	b2:13:ca:24:29:19	ARP	42	100.0.0.1 is at 8e:8c:af:6e:be:d0
4	0.412439	200.0.0.10	100.0.0.1	TCP	74	8087 → 33348 [SYN, ACK] Seq=0 Ack=1
5	0.412506	100.0.0.1	200.0.0.10	TCP	66	33348 → 8087 [ACK] Seq=1 Ack=1 Win=2
6	0.412663	100.0.0.1	200.100.0.15	TCP	74	42122 → 8089 [SYN] Seq=0 Win=29200 L
7	0.807138	200.100.0.15	100.0.0.1	TCP	74	8089 → 42122 [SYN, ACK] Seq=0 Ack=1
8	0.807199	100.0.0.1	200.100.0.15	TCP	66	42122 → 8089 [ACK] Seq=1 Ack=1 Win=2

Figure 14: Conexión TCP entre cliente-servidor

Desde el cliente se establecen dos conexiones, una hacia el servidor 1 cuya dirección IP es 200.0.0.10 en el puerto 8087 y la otra hacia el servidor 2 con dirección IP 200.100.0.15 en el puerto 8089. Esto se hace mediante la función ‘connect’ de la API de Sockets.

El cliente envía un mensaje hacia el servidor en el puerto indicado con la bandera SYN activada haciéndole saber al mismo que se desea conectar a él, posteriormente el servidor le responde con un mensaje con las banderas SYN y ACK activadas indicando la confirmación del paquete SYN enviado por el cliente.

Finalmente, para completar el establecimiento de la conexión, el cliente envía un mensaje al servidor con la bandera ACK activada indicando que ha recibido su mensaje

¹⁹La ip de R1, es 100.0.0.50, así está configurada en archivo IP_CONFIG, por lo cual tiene sentido la Figura13

con éxito. A partir de aquí se considera establecida la conexión TCP entre el cliente y el servidor. Esto sucede para ambos servidores, la diferencia es mediante los puertos que se comunican.

Notar que la conexión no resulta persistente, se cierra y abre en cada solicitud, es por ello que en cada pedido se establecerá un nuevo handshake entre client-server, según el método delegado a ejecutar remotamente.

4.1.2 Invocación y respuesta de procedimientos remotos

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	100.0.0.1	200.0.0.10	TCP	74	33348 → 8087 [SYN] Seq=0 Win=29200 L
2	0.365259	b2:13:ca:24:29:19	Broadcast	ARP	42	Who has 100.0.0.1? Tell 100.0.0.50
3	0.365328	8e:8c:af:6e:be:d0	b2:13:ca:24:29:19	ARP	42	100.0.0.1 is at 8e:8c:af:6e:be:d0
4	0.412439	200.0.0.10	100.0.0.1	TCP	74	8087 → 33348 [SYN, ACK] Seq=0 Ack=1
5	0.412506	100.0.0.1	200.0.0.10	TCP	66	33348 → 8087 [ACK] Seq=1 Ack=1 Win=2
6	0.412663	100.0.0.1	200.100.0.15	TCP	74	42122 → 8089 [SYN] Seq=0 Win=29200 L
7	0.807138	200.100.0.15	100.0.0.1	TCP	74	8089 → 42122 [SYN, ACK] Seq=0 Ack=1
8	0.807199	100.0.0.1	200.100.0.15	TCP	66	42122 → 8089 [ACK] Seq=1 Ack=1 Win=2

Figure 15: Captura de tráfico correspondiente a la ejecución del procedimiento remoto “suma”

A modo de ejemplo, analizaremos el procesamiento de la operación de suma entre los números 54 y 22.

Luego de establecida la conexión, se invoca desde el cliente el procedimiento que está implementado en el servidor. El cliente envía los datos a ser procesados por un procedimiento local en la biblioteca RPC, esta procesa los datos y los traduce al formato JSON-RPC 2.0 para enviarlos mediante la capa de transporte al servidor. La primera línea del tráfico capturado representa el envío de los datos hacia el servidor mediante un mensaje con la flag PSH activada, lo que le indica al receptor que los datos deben ser enviados a la aplicación tan pronto como sean recibidos.

0000	b2 13 ca 24 29 19 8e 8c af 6e be d0 08 00 45 00	...\$)...n...E
0010	00 75 7b 00 40 00 40 06 93 77 64 00 00 01 c8 00	u{.@.@.wd....
0020	00 0a 82 44 1f 97 f3 ff ae 2c 79 df 55 5e 80 18	...D....,y.U^..
0030	00 3a 2c 73 00 00 01 01 08 0a 00 1b 61 7b 00 1b	...s....a{...
0040	60 d9 7b 22 6a 73 6f 6e 72 70 63 22 3a 20 22 32	~{"json rpc": "2
0050	2e 30 22 2c 20 22 6d 65 74 68 6f 64 22 3a 20 22	.0", "me thod": "
0060	73 75 6d 61 22 2c 20 22 70 61 72 61 6d 73 22 3a	suma", " params":
0070	20 5b 35 34 2c 20 32 32 5d 2c 20 22 69 64 22 3a	[54, 22], "id":
0080	20 30 7d	0}

Figure 16: Contenido del paquete enviado por el cliente al llamar al procedimiento remoto “suma”

El servidor le comunica al cliente mediante un mensaje con la bandera ACK encen-

dida que ha recibido la solicitud con éxito. Luego recibe los datos, procesa la operación con los mismos y envía un mensaje con la respuesta y las banderas PSH y ACK encendidas, con el fin de que sean entregados al cliente al momento de ser recibidos.

The image shows a hex dump of a network packet. The left column contains hexadecimal addresses from 0000 to 0060 in increments of 10. The middle column contains the corresponding hexadecimal data. The right column contains the ASCII representation of the data. The data is a JSON object: {"json_rpc": "2", "result": 76, "id": 0}. The JSON is split across several lines in the dump.

Address	Hex Data	ASCII Data
0000	8e 8c af 6e be d0 b2 13 ca 24 29 19 08 00 45 00	...n... \$)...E
0010	00 5d 2a eb 40 00 3e 06 e5 a4 c8 00 00 0a 64 00	...]*.@.>...d
0020	00 01 1f 97 82 44 79 df 55 5e f3 ff ae 6d 80 18	...Dy U^...m
0030	00 39 68 3c 00 00 01 01 08 0a 00 1b 61 8d 00 1b	...9h<...a
0040	61 7b 7b 22 6a 73 6f 6e 72 70 63 22 3a 20 22 32	a{"json_rpc": "2
0050	2e 30 22 2c 20 22 72 65 73 75 6c 74 22 3a 20 37	.0", "result": 7
0060	36 2c 20 22 69 64 22 3a 20 30 7d	6, "id": 0}

Figure 17: Contenido del paquete enviado por el servidor con el resultado de la operación

El cliente le comunica al servidor que ha recibido el mensaje con el resultado. Posteriormente, desea cerrar la conexión con el servidor porque ha terminado de enviar los datos para la ejecución del procedimiento. Para esto envía un mensaje con la bandera FIN (finish) encendida indicando al receptor el cierre de la conexión TCP. Luego de esto, envía un nuevo mensaje de sincronización para establecer otra conexión con el servidor con el fin de ejecutar un nuevo procedimiento

Por otro lado, el servidor también le comunica al cliente que desea cerrar la conexión dado que no tiene más datos que procesar mediante un mensaje con la bandera FIN. El cliente recibe este mensaje y le responde al servidor con la confirmación de que recibió la solicitud de cierre de parte del servidor. El servidor le confirma al cliente mediante un mensaje con la confirmación del cierre de la conexión desde su lado. Cabe aclarar que el orden de procesamiento de estos mensajes puede variar dependiendo de si el receptor ha terminado de procesar los datos o no al momento de recibir la notificación de cierre por parte del emisor.

Por último, el servidor responde la solicitud de sincronización realizada por el cliente para iniciar una nueva conexión mediante un mensaje con las banderas SYN y ACK encendidas, para que luego el cliente confirme que ha recibido la respuesta exitosa del servidor para el establecimiento de la conexión.

El mismo análisis es válido para el tráfico capturado al realizarse una llamada a un procedimiento implementado por el servidor 2 desde el cliente. Las siguientes imágenes ilustran el tráfico capturado en la llamada al procedimiento “proc2” ubicado en el servidor 2, el cual nos devuelve el segundo parámetro de los dos que se le pasan. (Ignorar líneas 26 y 27).

25	1.197177	100.0.0.1	200.100.0.15	TCP	132	42122 → 8089	[PSH, ACK] Seq=1
26	1.238305	200.0.0.10	100.0.0.1	TCP	66	8087 → 33352	[FIN, ACK] Seq=43
27	1.238314	100.0.0.1	200.0.0.10	TCP	66	33352 → 8087	[ACK] Seq=77 Ack=4
28	1.354993	200.100.0.15	100.0.0.1	TCP	66	8089 → 42122	[ACK] Seq=1 Ack=67
29	1.355144	200.100.0.15	100.0.0.1	TCP	107	8089 → 42122	[PSH, ACK] Seq=1
30	1.355279	100.0.0.1	200.100.0.15	TCP	66	42122 → 8089	[ACK] Seq=67 Ack=4
31	1.355378	100.0.0.1	200.100.0.15	TCP	66	42122 → 8089	[FIN, ACK] Seq=67
32	1.355418	100.0.0.1	200.100.0.15	TCP	74	42126 → 8089	[SYN] Seq=0 Win=25
33	1.355559	200.100.0.15	100.0.0.1	TCP	66	8089 → 42122	[FIN, ACK] Seq=42
34	1.355563	100.0.0.1	200.100.0.15	TCP	66	42122 → 8089	[ACK] Seq=68 Ack=4
35	1.430355	200.0.0.10	100.0.0.1	TCP	66	8087 → 33352	[ACK] Seq=44 Ack=7
36	1.506742	200.100.0.15	100.0.0.1	TCP	66	8089 → 42122	[ACK] Seq=43 Ack=6
37	1.582698	200.100.0.15	100.0.0.1	TCP	74	8089 → 42126	[SYN, ACK] Seq=0

Figure 18: Captura de tráfico correspondiente a la ejecución del procedimiento remoto “proc2”

```

0000 b2 13 ca 24 29 19 8e 8c af 6e be d0 08 00 45 00 ...$)...n...E
0010 00 76 c2 f5 40 00 40 06 4b 18 64 00 00 01 c8 64 ...v.@.@K.d...d
0020 00 0f a4 8a 1f 99 b7 9a 19 b9 30 65 55 00 80 18 .....0eU...
0030 00 3a 2c dd 00 00 01 01 08 0a 00 1b 61 dc 00 1b ...;.....a...
0040 61 50 7b 22 6a 73 6f 6e 72 70 63 22 3a 20 22 32 aP{"json rpc": "2
0050 2e 30 22 2c 20 22 6d 65 74 68 6f 64 22 3a 20 22 .0", "me thod": "
0060 70 72 6f 63 32 22 2c 20 22 70 61 72 61 6d 73 22 proc2", "params"
0070 3a 20 5b 31 34 2c 20 32 32 5d 2c 20 22 69 64 22 : [14, 2 2], "id"
0080 3a 20 32 7d : 2}

```

Figure 19: Contenido del paquete enviado por el cliente al llamar al procedimiento remoto “proc2”

```

0000 8e 8c af 6e be d0 b2 13 ca 24 29 19 08 00 45 00 ...n....-$)...E
0010 00 5d 56 cc 40 00 3e 06 b9 5a c8 64 00 0f 64 00 ...]V.@.>..Z.d...d
0020 00 01 1f 99 a4 8a 30 65 55 00 b7 9a 19 fb 80 18 .....0e U.....
0030 00 39 63 79 00 00 01 01 08 0a 00 1b 61 f1 00 1b ...9cy.....a...
0040 61 dc 7b 22 6a 73 6f 6e 72 70 63 22 3a 20 22 32 a{"json rpc": "2
0050 2e 30 22 2c 20 22 72 65 73 75 6c 74 22 3a 20 32 .0", "re sult": 2
0060 32 2c 20 22 69 64 22 3a 20 32 7d 2, "id": 2}

```

Figure 20: Contenido del paquete enviado por el servidor con el resultado de la operación

4.1.3 Envío de notificaciones

Consiste en el envío de mensajes desde el emisor con el fin de solamente notificar al receptor, es decir, sin recibir ninguna respuesta de parte de este.

55	2.089355	100.0.0.1	200.100.0.15	TCP	74 42130 → 8089 [SYN] Seq=0 Win=2920
56	2.132162	200.100.0.15	100.0.0.1	TCP	66 8089 → 42128 [FIN, ACK] Seq=43 Ack=72
57	2.132171	100.0.0.1	200.100.0.15	TCP	66 42128 → 8089 [ACK] Seq=72 Ack=44 Win=0
58	2.215257	200.100.0.15	100.0.0.1	TCP	66 8089 → 42128 [ACK] Seq=44 Ack=72 Win=0
59	2.318985	200.100.0.15	100.0.0.1	TCP	74 8089 → 42130 [SYN, ACK] Seq=0 Ack=72
60	2.319055	100.0.0.1	200.100.0.15	TCP	66 42130 → 8089 [ACK] Seq=1 Ack=1 Win=0
61	2.319143	100.0.0.1	200.100.0.15	TCP	140 42130 → 8089 [PSH, ACK] Seq=1 Ack=1 Win=0
62	2.319160	100.0.0.1	200.100.0.15	TCP	66 42130 → 8089 [FIN, ACK] Seq=75 Ack=72
63	2.319230	100.0.0.1	200.0.0.10	TCP	74 33360 → 8087 [SYN] Seq=0 Win=2920
64	2.502490	200.0.0.10	100.0.0.1	TCP	74 8087 → 33360 [SYN, ACK] Seq=0 Ack=72
65	2.502809	100.0.0.1	200.0.0.10	TCP	66 33360 → 8087 [ACK] Seq=1 Ack=1 Win=0
66	2.502904	200.100.0.15	100.0.0.1	TCP	66 8089 → 42130 [ACK] Seq=1 Ack=75 Win=0

Figure 21: Captura de tráfico correspondiente al envío de una notificación

0000	b2 13 ca 24 29 19 8e 8c af 6e be d0 08 00 45 00	...\$) ... n...E
0010	00 7e d4 fe 40 00 40 06 39 07 64 00 00 01 c8 64	...@ @ 9 d...d
0020	00 0f a4 92 1f 99 3a 84 fb e9 3a 44 28 f8 80 18	...: ...:D(...
0030	00 3a 2c e5 00 00 01 01 08 0a 00 1b 62 f5 00 1b	...: , ... b ...
0040	62 da 7b 22 6a 73 6f 6e 72 70 63 22 3a 20 22 32	b {"json rpc": "2
0050	2e 30 22 2c 20 22 6d 65 74 68 6f 64 22 3a 20 22	.0", "me thod": "2
0060	4d 73 67 22 2c 20 22 70 61 72 61 6d 73 22 3a 20	Msg", "p arams":
0070	5b 22 48 65 6c 6c 6f 20 57 30 72 6c 64 22 5d 2c	["Hello W0rld"],
0080	20 22 69 64 22 3a 20 6e 75 6c 6c 7d	"id": n ull}

Figure 22: Contenido del paquete enviado por el cliente al enviar la notificación

Se observa que el campo “id”, que es el responsable de mapear correctamente los mensajes entre el receptor y emisor, en este caso está vacío. Esto debido a que como el receptor no debe enviar un mensaje de respuesta, no necesita saber el id del mensaje enviado. El comportamiento del tráfico es análogo a cuando se realiza la llamada a un procedimiento remoto desde el cliente, con la diferencia de que no existe una respuesta de parte del servidor (en los otros casos tenemos un mensaje hacia el cliente con la bandera PSH encendida indicando el resultado de la operación).

4.2 Análisis de captura de tráfico de los dos servidores

4.2.1 Establecimiento de la conexión

Con el fin de establecer la conexión entre el cliente y los servidores, 1 y 2, se comienza iniciando cada uno de los servidores, poniéndolos así en modo de escucha. A su vez, se establecen las direcciones IP y puertos correspondientes en los que deben escuchar.

Una vez los servidores se encuentran en modo de escucha se busca establecer la conexión por parte del cliente con ambos servidores, como se describió anteriormente.

A su vez, cuando los servidores se encuentran escuchando cada uno de estos queda en un bucle infinito que los mantiene en ejecución hasta detectarse alguna interrupción por parte del teclado (ctrl + c). Una vez interrumpida la ejecución se ejecuta un bloque

except, el cual cierra el servidor correspondiente y termina su ejecución de manera limpia a través de un `'sys.exit()'`.

1 0.000000	c6:af:11:77:27:7b	Broadcast	ARP	42 Who has 200.0.0.10? Tell 200.0.0.50
2 0.000009	d6:7f:50:03:98:d7	c6:af:11:77:27:7b	ARP	42 200.0.0.10 is at d6:7f:50:03:98:d7
3 0.038997	100.0.0.1	200.0.0.10	TCP	74 33348 → 8087 [SYN] Seq=0 Win=29200 Len=0 MSS=1460
4 0.039012	200.0.0.10	100.0.0.1	TCP	74 8087 → 33348 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0
5 0.399232	100.0.0.1	200.0.0.10	TCP	66 33348 → 8087 [ACK] Seq=1 Ack=1 Win=29696 Len=0 TS=

Figure 23: Conexión TCP entre servidor1-cliente

1 0.000000	ba:67:74:9b:2e:a8	Broadcast	ARP	42 Who has 200.100.0.15? Tell 200.100.0.50
2 0.000008	36:16:01:04:e3:84	ba:67:74:9b:2e:a8	ARP	42 200.100.0.15 is at 36:16:01:04:e3:84
3 0.039870	100.0.0.1	200.100.0.15	TCP	74 42122 → 8089 [SYN] Seq=0 Win=29200 Len=0 MSS=1460
4 0.039886	200.100.0.15	100.0.0.1	TCP	74 8089 → 42122 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0
5 0.286483	100.0.0.1	200.100.0.15	TCP	66 42122 → 8089 [ACK] Seq=1 Ack=1 Win=29696 Len=0 TS=

Figure 24: Conexión TCP entre servidor2-cliente

Cabe destacar que ambos servidores fueron modelados siguiendo el mismo diseño de código.

Con el fin de que los servidores puedan ejecutarse en paralelo se crea un hilo para cada uno, dónde se ejecutará el método `'serve()'`. Cada hilo fue configurado como `'daemon'` con el propósito de que el hilo siga corriendo aún cuando el programa haya finalizado.

4.2.2 Respuesta de procedimientos invocados por el cliente

6 0.762029	100.0.0.1	200.0.0.10	TCP	131 33348 → 8087 [PSH, ACK] Seq=1 Ack=1 Win=2
7 0.762180	200.0.0.10	100.0.0.1	TCP	66 8087 → 33348 [ACK] Seq=1 Ack=66 Win=29184
8 0.762301	200.0.0.10	100.0.0.1	TCP	107 8087 → 33348 [PSH, ACK] Seq=1 Ack=66 Win=
9 0.762314	200.0.0.10	100.0.0.1	TCP	66 8087 → 33348 [FIN, ACK] Seq=42 Ack=66 Win=
10 0.919283	100.0.0.1	200.0.0.10	TCP	66 33348 → 8087 [ACK] Seq=66 Ack=42 Win=2969
11 0.919448	100.0.0.1	200.0.0.10	TCP	66 33348 → 8087 [FIN, ACK] Seq=66 Ack=42 Win=
12 0.919453	200.0.0.10	100.0.0.1	TCP	66 8087 → 33348 [ACK] Seq=43 Ack=67 Win=2918

Figure 25: Captura de tráfico correspondiente a la ejecución del procedimiento remoto “suma”

Con el fin de visualizar lo que sucede con el tráfico cuando el cliente invoca un procedimiento remoto de algún servidor tomaremos al servidor 1 como ejemplo.

Vemos que el cliente, con dirección IP 100.0.0.1, le envía un segmento de datos al server1 con la bandera PSH encendida. El servidor le confirma la recepción de los datos al encender la bandera ACK y luego le envía la respuesta requerida con la bandera PSH encendida. Para finalizar la conexión el servidor 1 le envía una señal de FIN para cerrar

la transmisión de datos.

Por último, el cliente le confirma al servidor la recepción de datos con al encender la bandera ACK, después le envía también una señal de FIN para finalizar su parte de a conexión. El servidor culmina el intercambio confirmando la terminación encendiendo la bandera ACK.

4.2.3 Evaluación de las conexiones

A modo de visualización general de la conexión e intercambio de paquetes se utilizaron registros del servidor 1, pero reiteramos que la situación del servidor 2 es análoga.

Observamos que en ambas conexiones, cliente-server1 y cliente-server2, el protocolo TCP se utilizó adecuadamente para manejar el control de flujo; así como también para asegurar que los paquetes fuesen enviados, recibidos y confirmados. En ningún momento se observan retransmisiones ni errores evidentes en el intercambio, lo que nos indica que la comunicación fue exitosa y eficiente para todas las operaciones ejecutadas. En todo caso si así fuese TCP por naturaleza corregiría el envío de paquetes, por otro lado al capturar el flujo de los routers, no se han notado diferencias, es decir, no se vislumbra nada particular, incluso la ip de los routers no aparece, más allá del inicio en el protocolo ARP.

5 Conclusión

Se logró cumplir de manera efectiva con todos los requerimientos del proyecto, lo cual ha permitido adquirir conocimientos sobre redes de computadoras.

A lo largo de este proceso, nos familiarizamos con conceptos esenciales de redes e Internet, utilizando herramientas como Wireshark, tcpdump, traceroute y ping para diagnosticar y depurar el comportamiento de la red. El análisis de tráfico nos ha permitido entender cómo la latencia y la pérdida de paquetes varían según factores como la configuración del servidor y la topología de la red.

Se pudo observar como operan diversos protocolos HTTP, HTTPS, DNS, ICMP, TCP, etc. Y nos da la idea de la infraestructura global de internet, construida por diversas capas, con operaciones y responsabilidades definidas.

Pudimos realizar una aplicación siguiendo la arquitectura cliente-servidor, mediante el estándar establecido por json_rpc 2.0, y montarla en una red emulada por Mininet. Donde pudimos ver e implementar el funcionamiento de sockets. Para posteriormente analizar dicha red mediante captura de tráfico en las interfaces emuladas.

No solo hemos adquirido comprensión práctica del funcionamiento de las redes, sino que también ha fortalecido nuestras habilidades en el manejo de herramientas de análisis y la aplicación de arquitecturas modernas en entornos simulados.

6 Anexos

6.1 capturas de casos de pruebas

```
EXIT0: Todas las pruebas completadas.  
mininet> client python3.8 clientTest.py & python3.8 clientTest.py & python3.8  
clientTest.py & python3.8 clientTest.py  
[1] 2610  
[2] 2611  
[3] 2612  
Iniciando pruebas de casos sin errores.  
Iniciando pruebas de casos sin errores.  
Iniciando pruebas de casos sin errores.  
prueba de test simple completada  
Iniciando pruebas de casos sin errores.  
prueba de test simple completada  
prueba de test simple completada  
prueba de test simple completada  
prueba de test simple completada  
prueba de test simple completada  
prueba de test simple completada  
prueba de test simple completada  
prueba de test simple completada  
prueba de test simple completada  
prueba de test simple completada  
prueba de test simple completada  
prueba de test simple completada  
prueba de test simple completada  
prueba de test simple completada  
prueba de test simple completada  
prueba de test notificacion completada  
prueba de test notificacion completada  
prueba de test simple completada  
prueba de test notificacion completada  
prueba de test notificacion completada  
prueba de test notificacion completada  
prueba de test notificacion completada  
prueba de test notificacion completada  
prueba de test notificacion completada  
prueba de test notificacion completada  
prueba de test notificacion completada  
prueba de test notificacion completada  
prueba de test notificacion completada  
prueba de test notificacion completada  
prueba de test notificacion completada  
prueba de test notificacion completada  
prueba de test notificacion completada  
prueba de test metodo params opcionales completada  
prueba de test metodo params opcionales completada  
prueba de test metodo params opcionales completada
```

Figure 26: Pruebas en mininet multipleClient

```

prueba de test metodo params opcionales completada
prueba de test metodo params opcionales completada
prueba de test metodo params opcionales completada
prueba de test metodo params opcionales completada
prueba de test metodo params opcionales completada
prueba de test metodo params opcionales completada
prueba de test metodo params opcionales completada
prueba de test metodo params opcionales completada
prueba de test metodo params opcionales completada
prueba de test metodo params opcionales completada
prueba de test metodo que retorna lista completada
prueba de test metodo que retorna lista completada
prueba de test metodo que retorna lista completada
prueba de test metodo sin parametros completada
=====

ÉXITO: Pruebas de casos sin errores completadas.
=====

Iniciando pruebas de casos con errores.
prueba de test metodo sin parametros completada
=====

ÉXITO: Pruebas de casos sin errores completadas.
=====

Iniciando pruebas de casos con errores.
prueba de test metodo sin parametros completada
=====

ÉXITO: Pruebas de casos sin errores completadas.
=====

Iniciando pruebas de casos con errores.
prueba de test metodo sin parametros completada
=====

ÉXITO: Pruebas de casos sin errores completadas.
=====

Iniciando pruebas de casos con errores.
Lanzó excepcion, metodo sin definir: {'code': -32601, 'message': 'Procedimiento
no encontrado'}
Lanzó excepcion, metodo sin definir: {'code': -32601, 'message': 'Procedimiento
no encontrado'}
Lanzó excepcion, metodo sin definir: {'code': -32601, 'message': 'Procedimiento
no encontrado'}
Lanzó excepcion, Error de parseo : {code: -32700, message: Error de parseo}
Lanzó excepcion, Error de parseo : {code: -32700, message: Error de parseo}
Lanzó excepcion, Error de parseo : {code: -32700, message: Error de parseo}
Lanzó excepcion, metodo sin definir: {'code': -32601, 'message': 'Procedimiento
no encontrado'}

```

Figure 27: Pruebas en mininet multipleClient


```

=====
ÉXITO: Pruebas de casos con errores completadas.
=====

Iniciando prueba servidor ajeno
Lanzó excepcion, Error Interno: {'code': -32603, 'message': 'Error Interno'}
=====

ÉXITO: Pruebas de casos con errores completadas.
=====

Iniciando prueba servidor ajeno
Lanzó excepcion, Error Interno: {'code': -32603, 'message': 'Error Interno'}
=====

ÉXITO: Pruebas de casos con errores completadas.
=====

Iniciando prueba servidor ajeno
Test simple completado.
Lanzó excepcion, Error Interno: {'code': -32603, 'message': 'Error Interno'}
=====

ÉXITO: Pruebas de casos con errores completadas.
=====

Iniciando prueba servidor ajeno
Test simple completado.
Test simple completado.
Test de notificación completado.
Test de notificación completado.
Test de notificación completado.
Test simple completado.
Test de notificación completado.
Test de múltiples parámetros completado
Test de múltiples parámetros completado
Test de múltiples parámetros completado
Test de múltiples parámetros completado
Test de suma completado.
Test de suma completado.
Test de suma completado.
Test de suma completado.
Segundo test de suma con 10 parámetros completado
Iniciando pruebas de errores, servidor ajeno
Segundo test de suma con 10 parámetros completado
Iniciando pruebas de errores, servidor ajeno
Segundo test de suma con 10 parámetros completado
Iniciando pruebas de errores, servidor ajeno
Segundo test de suma con 10 parámetros completado
Iniciando pruebas de errores, servidor ajeno
Llamada incorrecta sin parámetros. Genera excepción necesaria.
{'message': 'Internal error.', 'code': -32603}
Llamada incorrecta sin parámetros. Genera excepción necesaria.
{'message': 'Internal error.', 'code': -32603}
Llamada incorrecta sin parámetros. Genera excepción necesaria.
{'message': 'Internal error.', 'code': -32603}
Llamada incorrecta sin parámetros. Genera excepción necesaria.
{'message': 'Internal error.', 'code': -32603}

```

Figure 28: Pruebas en mininet multipleClient

```

EXIT0: Todas las pruebas completadas.
mininet> client python3.8 clientTest.py & python3.8 clientTest.py & python3.8
clientTest.py & python3.8 clientTest.py
Llamada incorrecta genera excepción interna del servidor.
{'message': 'Internal error.', 'code': -32603}
=====

EXIT0: Todas las pruebas completadas.
Llamada incorrecta genera excepción interna del servidor.
{'message': 'Internal error.', 'code': -32603}
=====

EXIT0: Todas las pruebas completadas.
Llamada incorrecta genera excepción interna del servidor.
{'message': 'Internal error.', 'code': -32603}
=====

EXIT0: Todas las pruebas completadas.
[4] 2726
[5] 2728
[6] 2729
Iniciando pruebas de casos sin errores.
Iniciando pruebas de casos sin errores.
Iniciando pruebas de casos sin errores.
Iniciando pruebas de casos sin errores.
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test simple completada
prueba de test notificacion completada
prueba de test notificacion completada
prueba de test notificacion completada
prueba de test notificacion completada
ERROR durante la ejecución de pruebas: [Errno 111] Connection refused
ERROR durante la ejecución de pruebas: [Errno 111] Connection refused
ERROR durante la ejecución de pruebas: [Errno 111] Connection refused
ERROR durante la ejecución de pruebas: [Errno 111] Connection refused
mininet> █

```

Figure 29: Pruebas en mininet finalizadas y reejecución multiple Clients cortando conexión de server en medio del proceso

Al ser la ejecución concurrente se nota que al finalizar el cliente se habilita el prompt de la terminal y luego de ingresado el comando se muestra el fin de las ejecuciones clientes faltantes

References

- [Akamai] Akamai Technologies. (2024) ¿Qué es un ataque DDoS de inundación ICMP? *Akamai - Seguridad*. Recuperado el 25 de agosto de 2024, de <https://www.akamai.com/es/glossary/what-is-icmp-flood-ddos-attack>
- [Redes de computadoras: Un enfoque descendente] Kurose, J. F. y Ross, K. W. (2017). *Redes de computadoras: Un enfoque descendente*. Pearson Educación.
- [RFC 826, 1982] Plummer, D. C. (1982). An Ethernet Address Resolution Protocol. Recuperado el 9 de septiembre de 2024, de <https://www.rfc-editor.org/rfc/rfc826>
- [DNS AAAA/A] Cao, D. (2023). Registros DNS AAAA. *HowToUseLinux*. Recuperado el 12 de agosto de 2024, de <https://www.howtouselinux.com/post/dns-aaaa-records>
- [Hostwinds] Hostwinds Team. (2021) ¿Qué son los encabezados IP? *Hostwinds Tutoriales*. Recuperado el 15 de agosto de 2024, de <https://www.hostwinds.es/tutorials/what-are-ip-headers>
- [JSON RPC 2.0, 2013] JSON RPC Working Group. (2013). JSON RPC 2.0 Specification. *JSON-RPC*. Recuperado el 21 de agosto de 2024, de <https://www.jsonrpc.org/specification>
- [MDN Web Docs, 2024] MDN Web Docs. (2024). Content-Encoding. *MDN Web Docs*. Recuperado el 14 de agosto de 2024, de <https://developer.mozilla.org/es/docs/Web/HTTP/Headers/Content-Encoding>
- [Paessler, the monitoring experts, 2024] Paessler, the monitoring experts. (2024). Ping. *Paessler, the monitoring experts..* Recuperado el 15 de agosto de 2024, de <https://www.paessler.com/es/it-explained/ping#:~:text=Packet%20InterNet%20Groper.-,%C2%BFC%C3%B3mo%20funciona%20Ping%3F,eco%20a%20la%20direcci%C3%B3n%20indicada.>
- [Sockets Python, 2024] McMillan, G. (2024). HOW TO - Programación con sockets. *Documentación de Python - 3.12.5*. Recuperado el 22 de agosto de 2024, de <https://docs.python.org/es/3/howto/sockets.html>
- [TCPDUMP, 2024] TCPDUMP. (2024). tcpdump(1) - Linux man page. *tcpdump.org*. Recuperado el 12 de agosto de 2024, de <https://www.tcpdump.org/manpages/tcpdump.1.html>

- [Wikipedia, la enciclopedia libre, 2024] Wikipedia, la enciclopedia libre. (2024) Cabecera IP. *Wikipedia, la enciclopedia libre*. Recuperado el 15 de agosto de 2024, de https://es.wikipedia.org/wiki/Cabecera_IP#cite_note-1
- [Wikipedia, la enciclopedia libre, 2024] Wikipedia, la enciclopedia libre. (2024) Protocolo de control de mensajes de Internet. *Wikipedia, la enciclopedia libre*. Recuperado el 15 de agosto de 2024, de [https://es.wikipedia.org/wiki/Protocolo_de_control_de_mensajes_de_Internet#:~:text=El%20Echo%20Request%20\(Petici%C3%B3n%20eco,protocolo%20ICMP%2C%20subprotocolo%20de%20IP](https://es.wikipedia.org/wiki/Protocolo_de_control_de_mensajes_de_Internet#:~:text=El%20Echo%20Request%20(Petici%C3%B3n%20eco,protocolo%20ICMP%2C%20subprotocolo%20de%20IP)
- [Wireshark User's Guide, 2024] Sharpe, R., Warnicke, E., y Lamping, U. (2024). Wireshark User's Guide, Versión 4.3.2. *Wireshark*. Recuperado el 12 de agosto de 2024, de https://www.wireshark.org/docs/wsug_html_chunked/
- [Traceroute Using an IP Option. RFC 1393] Malkin, G. (1993). Traceroute Using an IP Option. *RFC 1393*. Recuperado el 16 de agosto de 2024, de <https://www.rfc-editor.org/pdf/rfc/rfc1393>.