Informe Obligatorio de Redes de Computadoras

Matias Rugnon, German Capurro, Tiago Calero

12 de septiembre de 2025

Índice

1. Introducción

Este documento tiene como fin presentar los resultados de las actividades exploratorias provistas por el equipo docente, en estas se buscó comprender las distintas herramientas que serán útiles para el curso

2. Conceptos Teóricos

- Capas del modelo OSI y TCP/IP.
- Protocolos estudiados (HTTP, TCP, IP, etc.).

Interfaz Loopback:
Se trata de una interfaz interna del sistema, es decir, una interfaz que representa una tarjeta de red ficticia, en la que solo existe tráfico interno del equipo. Sirve para poder crear una conexión cliente-servidor interna en el equipo.

3. Herramientas y comandos

Se presentaron diferentes herramientas de trabajo como:

- TCPDUMP
- WireShark
- TShark
- Comando ping
- Traceroute
- Telnet

4. Presentación de Herramientas y comandos

En esta parte se nos pidió investigar las herramientas presentadas por los docentes (TCPDump, WireShark y TShark), buscando sus similitudes y diferencias a la hora de su uso.

1. TCPDump

TCPDump es una herramienta que se utiliza para capturar el tráfico de manera sencilla y sin interfaz gráfica sobre la terminal de LINUX/UNIX, (notar que no está disponible de manera nativa en Windows).Como fue mencionado, opera sobre la terminal de Linux. La herramienta permite capturar el tráfico de paquetes por la red, y permite filtrar el tráfico por host, puerto, tráfico por TCP, por UDP, entre otros.
Personalmente, esta herramienta es la que encontramos más cómoda y sencilla de utilizar debido a la fácil tipografía de comandos, simpleza al momento de capturar tráfico, y la facilidad de acceder a información para aprender su uso, sea su página web o el apoyo

de herramientas de inteligencia artificial para acceso rápido a preguntas sobre su uso más específicas.
Su función más útil fue la creación de archivos .pcap para permitirnos analizar en otra herramienta el tráfico capturado más tarde.
Utiliza filtros BPF (Berkley Packet Filter), diferentes a los filtros utilizados por Wireshark y TShark. La característica de estos filtros es que se aplican antes de comenzar la captura. Un ejemplo de estos filtros es:

```
tcp port 80
```

2. WireShark

WireShark ofrece una interfaz gráfica amigable para poder capturar paquetes, interfaz que no ofrece ni TCPDump ni TShark, que funcionan en la línea de comando. Wireshark también se diferencia en que presenta diferentes métodos de análisis de los paquetes capturados, brindando estadísticas y gráficos de los mismos (como qué porcentaje fueron TCP, cuáles UDP, entre otras funcionalidades). Notar también que esta herramienta esta disponible en Windows .
Como mencionamos anteriormente, TCPDump la consideramos a la herramienta de captura de trafico mas amigable y sencilla de usar, pero vale recalcar que si bien usamos TCPDump para capturar el tráfico WireShark nos permitió analizarlo de forma sencilla y ordenada gracias a la interfaz user-friendly.
Los filtos en WireShark (al igual que en TShark) son filtros de visualización y estos se aplican mediante captura o luego de la misma. Un ejemplo de este filtro es

```
tcp.port == 80
```

3. TShark

TShark ofrece funcionalidades similares a WireShark, pero sin interfaz gráfica (GUI), y con salida en consola, al igual que TCPDump.
Aunque su formato de salida es menos visual e interactivo, lo que puede dificultar el análisis manual detallado, es mucho más liviano y eficiente para procesar grandes volúmenes de datos o para integrarse en scripts de automatización y entornos sin interfaz gráfica. Se podría decir que es un equivalente a TCPDump pero el cual funciona en Windows, si bien no fue la herramienta que mas usamos luego de su investigación consideramos que puede hacer todo lo que hacen las otras herramientas, tanto filtrar por flags y guardar lo capturado.

4. Comando ping

La principal función del comando ping es obtener información de red entre dos dispositivos(la computadora en la que se ejecuta y otro dispositivo, sea un servidor, otra computadora, página web, etc). La información de conexión que se obtiene se realiza enviando paquetes de prueba los cuales generan un retorno por parte del destinatario. Esta

información consta de, primero y más importante si el paquete llegó, en qué tiempo lo hizo y si hay posibles pérdidas de paquetes.
En conclusión, su función es, decir si hay conexión y qué tan rápida y confiable es.

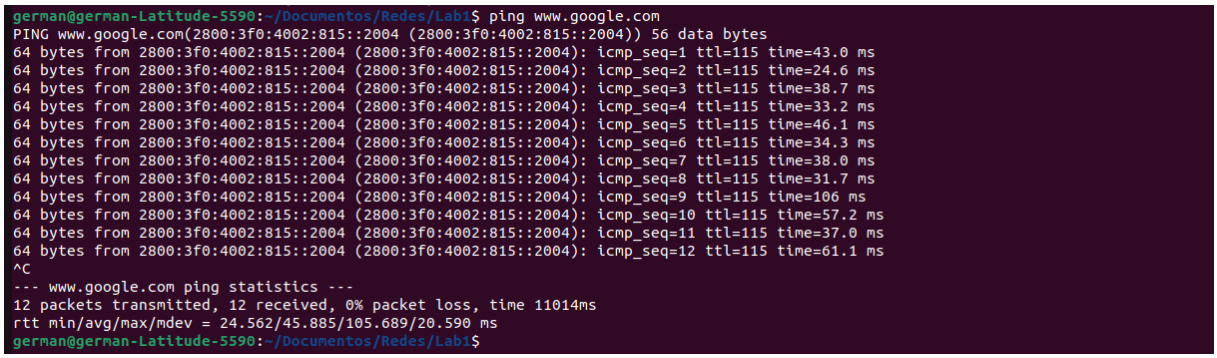

Figura 1: Aquí hay un ejemplo al momento de uso del comando al conectar con www.google.com

5. Traceroute

La herramienta traceroute tiene como su función brindar un método de analizar una ruta de conexión de red. A diferencia del comando ping el cual obtiene datos de la conexión, traceroute obtiene el camino de conexión, es decir muestra todos los puntos medios de la conexión(nodos de red). Esto se obtiene enviando una sucesión de paquetes que, al llegar a un nuevo nodo, retornan los datos correspondientes y luego se envía otro, repitiendo el proceso hasta llegar al destino.
Estos datos pueden usarse para ver por donde pasa la conexión, determinar en donde se rompe en caso de que no exista la conexión o hallar un nodo con 'cuello de botella' que retarda la conexión.

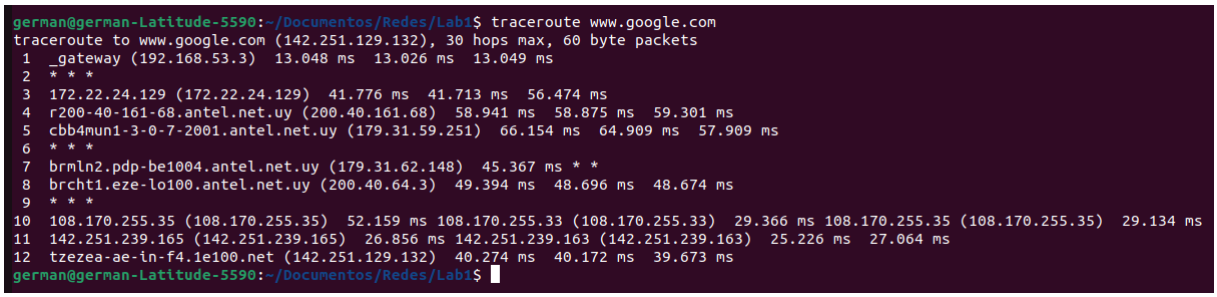

Figura 2: Aquí un ejemplo del uso de traceroute al establecer una conexión con www.google.com

Notar en la Figura 2 las lineas que indican los caracteres "* * *", esto indica que el router/nodo no dio respuesta.

5. Actividad Exploratoria - Parte 1

La parte de investigación se encuentra en la sección "Presentación de Herramientas y Comandos "(pg.3)

5.1. Parte 2

En esta parte se buscó capturar tráfico de paquetes con las herramientas presentadas en la parte anterior (ver sección "presentación de Herramientas y comandos").
En estas capturas se buscó encontrar los distintos protocolos estudiados durante el curso y poder ver de manera práctica su funcionamiento.

1. Captura de trafico por interfaz Loopback
En esta parte se nos pidió utilizar dos terminales para cumplir los roles de cliente, utilizando el comando telnet para abrir una conexión local del servicio, y servidor, además, debimos utilizar una tercera terminal para realizar la captura en el loopback y guardarla para posterior análisis. Para esta última terminal en particular ejecutamos el comando 'sudo tcpdump -i lo host 127.0.0.1 and port 5001 -w incognito.pcap' en una terminal bash de linux-ubuntu, el cual nos permitió mediante tcpdump capturar el tráfico de la ip host 127.0.0.1(la cual es una ip local, es decir no nos conectamos a ningún otro dispositivo), el puerto 5001 y guardarlo en el archivo incognito.pcap para luego poder analizarlo en WireShark.

El servidor se levantó utilizando el archivo "servidor-incognito.py", luego, se abrió una sesión con el servidor usando el comando telnet en el loopback.

2. Análisis de la captura
Con las ejecuciones de las lineas en el lado del cliente, descubrimos que el servidor funciona como un "generador de citas de grandes personajes de la historia mundial", ya que cada vez que hacíamos una solicitud, el servidor devolvía de manera aparentemente aleatoria una frase célebre. Unos ejemplos son

```
La felicidad no es algo hecho. Proviene de tus propias acciones.
- Dalai Lama

No cuentes los dias, haz que los dias cuenten.
- Muhammad Ali
```

Notar que tanto el comando afzx como el bb aparentemente devolvían los dos lo mismo, es decir, una frase aleatoria.

Identificamos también que el comando affhex generaba una respuesta con la fecha y hora actual/del sistema, el comando mas claro fue el final, zzz, el cual claramente cerraba la conexión cliente servidor.

Todos estos análisis fueron posibles de realizar con mayor detenimiento gracias a haber capturado el trafico en el archivo incognito.pcap el cual al introducirlo a WireShark pudimos usar al función de la herramienta para seguir el flujo tcp y leer el intercambio de mensajes, dado que los mismos no estaban cifrados.

La herramienta "Following Protocol Streams"de WireShark, nos mostró el intercambio

de solicitudes-respuestas que tuvo la interacción del cliente con el servidor montado.

–adjuntar captura––

5.2. Parte 3

El objetivo de esta parte fue aprender a visualizar capturas de tráfico no aisladas, como así fue en la Parte 1 donde se capturo unicamente trafico del local host. En esta parte se capturó el tráfico(sin filtos específicos en este caso lo razón por la cual mencionaremos mas adelante) mientas ingresábamos a dos links url mediante un navegador web para poder capturar y analizar intercamos de mensajes por protocolo HTTP y HTTPS, las direcciones a las que ingresamos fueron:

- `https://www.fing.edu.uy`
- `http://www.columbia.edu/~fdc/family/ssamerica2.jpg`

1. Captura https://www.fing.edu.uy

Como mencionamos anteriormente la capturas en esta instancia las realizamos sin aplicar filtros en tcpdump mas haya de las flags `-i wlp2s0` y `-w https-web.pcap` las cuales nos permitieron filtrar por conexión wifi y guardarlas en un archivo para analizar detenidamente en WireShark.

Al comenzar nuestro análisis de la captura lo primero que encontramos(relacionado a nuestro objetivo de captura) fue la comunicación DNS, tal y como vimos en el curso la función de la misma es obtener la dirección ip del servidor para lograr realizar la conexión. Por lo general al iniciar una conexión vía internet, en este caso mediante el protocolo https, el cliente no conoce la dirección ip del servicio, por lo que siempre antes de iniciar las conexiones el mismo la obtiene mediante una consulta DNS. Esta consiste en (a grandes rasgos) enviar una solicitud "`standard query`.[a] un servidor DNS local conocido el cual logra conocerla dirección ip a la que corresponde el dominio buscado al realizar y preguntar a otros servidores DNS y llegar al servidor DNS autoritativo(a quien se refiere el dominio), este responde con una "`standard query response`"la cual indica su ip y el servidor DNS local se lo comunica al host solicitante. En nuestro caso la ip retornada por la consulta al dominio `www.fing.edu.uy` fue la dirección: `164.73.32.20`.

Luego filtramos en la linea de comandos de Wireshark para obtener el flujo de llegada de los paquetes mediante el filtro `ip.src == 164.73.32.20 and tcp.port == 443`. Pero al seguir el flujo no nos es posible leer la consulta. Esto ocurre ya que el protocolo de capa de aplicación utilizado sobre tcp en este caso no es http sino https, la diferencia reside en que https(HyperText Transfer Protocol Secure) es, como lo indica su nombre, seguro. Para esto se utiliza un cifrado de tal manera que los únicos que pueden leerlo y descifrarlo son el servidor de origen y el cliente, lo que permite que no pueden ser leídos ni modificados por alguien externo que capture o detecte los paquetes. La razón por la cual es posible leer los mensajes con protocolo http es gracias a que los mensajes se transmiten en texto plano de ASCII 7bits.

2. Captura http://www.columbia.edu/ fdc/family/ssamerica2.jpg

Como mencionamos, la mayor diferencia entre los protocolos http y https, es la encriptación. En esta captura, el protocolo utilizado para obtener la conexión mediante un link url fue http, por lo que es posible analizar a fondo todos los paquetes capturados. Sin embargo tuvimos inconvenientes que consideramos que si bien no vienen al contenido del curso vale la pena mencionar. Hasta el momento el ecosistema utilizado para realizar todas las capturas fue una laptop linux-ubuntu con navegador Firefox, lo que no nos permitía captar los paquetes esperados, esto ocurría debido a que por defecto el navegador convierte a https toda consulta http a la que se ingrese presionando un link, luego de desactivar la función nos encontramos con que no lográbamos capturar el paquete http respuesta 200 OK, problema que no supimos solucionar por lo que hicimos un cambio de ecosistema para esta captura.

Al inicio de la captura volvemos a observar el intercambio de mensajes DNS para obtener la ip address del dominio. Pocos paquetes después nos encontramos con el primer paquete http. El primer paquete capturado fue el mensaje de solicitud http,conteniendo el cabezal `GET / fdc/family/ssamerica2.jpg HTTP/1.1`, que como lo dice su nombre, solicita los contenidos del archivo ssamerica2.jpg, el campo `Host` que indica a qué servidor se Mozilla 5.0 y el `Accept-Encoding y Accept-Language` que indican qué tipo de archivos y lenguajes aceptamos, en nuestro caso se aceptan archivos html, xhtml e imágenes, aceptamos únicamente el lenguaje español(la versión en español, si es que existe).

Luego de obtener muchos paquetes con de protocolo tcp observamos un paquete con protocolo http con cabezal `HTTP/1.1 200 OK (image/jpeg)`, este paquete nos indica dos campos adicionales de gran importancia, `Content-Type y Content-Length`, estos nos indican el tipo de archivo que vamos a recibir, en este caso una imagen jpeg y que tamaño tiene en bytes, en nuestro caso 222449B, aprox 222.5KB.

También observamos el campo `last-modified` en el segundo paquete, su función no le compete al cliente sino a un servidor proxy el cual cachea el archivo en caso de volver a recibir una solicitud para el mismo dominio. En caso de recibirlo, este solicita únicamente la última fecha de modificación al servidor del dominio para verificar que tenga el archivo actualizado, esto se realiza para lograr menores tiempos de respuesta.

6. Documentación de la Biblioteca `xmlrpc_redes`

6.1. Visión general de la biblioteca

En esta sección se documenta la biblioteca desarrollada para la Parte 2 del obligatorio. Se explican las funciones implementadas, el diseño de la arquitectura cliente-servidor, las decisiones de diseño adoptadas y cómo se relacionan con los conceptos vistos en el curso de Redes de Computadoras.
La biblioteca `xmlrpc_redes` implementa el protocolo XML-RPC sobre sockets TCP, siguiendo la especificación provista en el obligatorio. Proporciona una arquitectura cliente-servidor con stubs para la serialización (marshalling) y deserialización (unmarshalling) de parámetros.

6.2. Archivos de la librería

Para la implementación de la librería se incluyeron cuatro archivos

- `xmlrpc_redes.py`
- `server.py`
- `client.py`
- `test_app.py`

6.3. Comportamiento del cliente

El archivo que implementa el cliente define el flujo de interacción en el modelo XML-RPC. Su funcionamiento consiste en construir el llamado XML (*marshalling*), encapsularlo en una solicitud HTTP/1.1 de tipo `POST`, y luego conectarse al servidor mediante un socket TCP para enviar la operación junto con los parámetros.

A continuación, el cliente:

1. Construye el mensaje XML con el procedimiento remoto y sus parámetros.
2. Encapsula el mensaje en una solicitud HTTP.
3. Abre un socket TCP hacia la dirección del servidor.
4. Envía la solicitud.
5. Recibe la respuesta desde el servidor.
6. Convierte la respuesta XML en tipos nativos de Python (*unmarshalling*).
7. Visualiza el resultado final al usuario.

6.4. Comportamiento del Servidor

El archivo que implementa el servidor sigue un comportamiento análogo, pero en este caso inicia escuchando y aceptando conexiones entrantes. Cada vez que un cliente se conecta, el servidor crea un hilo independiente y establece una conexión TCP dedicada para atender las solicitudes de dicho cliente.

El flujo de ejecución del servidor es el siguiente:

1. Inicializa un socket TCP en la dirección y puerto configurados.
2. Queda en estado de escucha de conexiones entrantes.
3. Acepta una conexión cuando un cliente solicita conectarse.
4. Crea un nuevo hilo (`threading.Thread`) para atender a ese cliente.
5. Dentro del hilo:

- Recibe y parsea la solicitud HTTP.
- Valida encabezados y el método `POST`.
- Extrae y parsea el cuerpo XML-RPC.
- Invoca el procedimiento solicitado si existe.
- Construye la respuesta HTTP con el resultado o un error.
- Envía la respuesta al cliente.

6.5. Tecnologías utilizadas

Para poder implementar la arquitectura cliente-servidor se utilizaron diferentes librerías y tecnologías del lenguaje Python, entre ellas se encuentran:

- **ElementTree (módulo `xml.etree`)**: se utilizó para serializar y deserializar mensajes XML, siguiendo el estándar de XML-RPC. Se eligió por ser parte de la librería estándar de Python y permitir un manejo directo de estructuras XML.
- **Sockets**: se emplearon *sockets* TCP (API POSIX) para la comunicación cliente-servidor. El protocolo TCP fue elegido por su confiabilidad en la entrega de mensajes, ya que garantiza la transmisión ordenada, sin pérdidas y sin duplicados. La biblioteca implementada utiliza las funcionalidades provistas por la API de *sockets* para la creación, configuración y manejo de conexiones de red, pudiendo extenderse tanto a TCP como a UDP.
- **Threading**: se empleó la librería `threading` de Python para manejar múltiples conexiones de clientes de forma concurrente. Cada vez que el servidor acepta una nueva conexión, crea un hilo independiente encargado de procesar las solicitudes de ese cliente, permitiendo que el servidor atienda a varios clientes de manera simultánea sin bloquear la ejecución principal.

6.6. Protocolos

Se siguió estrictamente el estándar **XML-RPC** y el protocolo **HTTP/1.1**, utilizando unicamente solicitudes de tipo `POST` como se especifica en el protocolo **XML-RPC**.

6.7. Documentación de la API de la biblioteca

6.7.1. Funciones principales de `XMLRPC_redes`

- `serialización(valor)`: traduce un valor de un tipo de python a un elemento xml del mismo tipo con el tag 'value' y con el mismo valor, usado posteriormente en los mensajes del protocolo `XML-RPC`.
- `deserialización(elem)`: traduce un elemento xml de tag 'value' a un valor de python del tipo correspondiente y mismo valor, usado posteriormente en los mensajes del protocolo `XML-RPC`.
- `construir_llamado_xml(method, params)`: construye el cuerpo xml del llamado realizado por el cliente, recibe el nombre del método y sus parámetros, luego retorna el llamado xml del método según el protocolo XML-RPC.

- `parsear_llamado_xml(xml)`: recibe un cuerpo xml construido por la función anterior y luego retorna el nombre del método recibido junto a sus parámetros.
- `construir_respuesta_xml(result)`: recibe el resultado obtenido de la ejecución del método y luego construye el cuerpo xml de la respuesta al llamado del método según el protocolo XML-RPC.
- `construir_error_xml(err, msg)`: se utiliza cuando surge un error luego de recibir el llamado XML-RPC, recibe el código del error junto al mensaje del error y luego construye la respuesta de error en xml según el protocolo XML-RPC y la retorna.
- `parsear_respuesta_xml(xml)`: se utiliza para leer la respuesta xml obtenida del servidor, determina si el llamado fue exitoso o si retorno un error, devuelve un booleano indicando el caso y el valor del resultado si fue exitoso.
- `construir_llamado_http(host, data)`: construye el llamado xml, recibe el host y el cuerpo xml, crea los encabezados http según la especificación del protocolo XML-RPC y los retorna junto al cuerpo xml listo para enviar.
- `parsear_llamado_http(resp)`: se utiliza para leer el llamado XML-RPC, recibe el llamado y retorna la linea de llamado, los encabezados http y el cuerpo xml.
- `construir_respuesta_http(data)`: construye la respuesta http al llamado, recibe el cuerpo xml, crea los encabezados de respuesta http especificados en el protocolo XML-RPC y los junta con el cuerpo xml para retornar la respuesta, pronta para enviar.
- `parsear_respuesta_http(resp)`: ídem a parsear_llamado_http pero ejecutada por el cliente en vez del servidor para leer la respuesta al llamado http.

6.7.2. Funciones principales de `Server`

- `__init__(address)`: inicializa los datos del servidor, recibe una dupla con la dirección ip y un puerto.
- `add_method(func)`: registra un procedimiento remoto. El nombre utilizado será `func.__name__`.
- `serve()`: inicia el servidor, abre el socket y queda a la espera de conexiones. Cada cliente es atendido en un hilo independiente.
- `atender_cliente(conn, peer)`: procesa la solicitud de un cliente. Incluye:
  - Parseo de la solicitud HTTP recibida.
  - Verificación de encabezados HTTP obligatorios (`Host`, `Content-Type`, `Content-Length`).
  - Validación de que el método HTTP sea `POST`.
  - Parseo del cuerpo XML-RPC con ElementTree.
  - Ejecución del método invocado y construcción de la respuesta XML.
  - Manejo de errores y envío de respuestas `fault`.

- `error(conn, num_err, mensaje)`: construye una respuesta de error XML-RPC con un código de error y un mensaje descriptivo.

6.7.3. Funciones principales de `Client`

- `__init__(address, port, timeout)`: inicializa los datos del cliente, recibe la dirección ip, el puerto y un timeout en segundos.
- `__getattr__(method_name)`: se ejecuta para realizar la invocación del método remoto. Cuando se ejecuta el método desde el cliente, al este estar definido remotamente, no se encontrara en la clase Client por lo que se ejecutara esta función automáticamente con el nombre del método por parámetro, luego se invoca la función `_invoke`, con el nombre del método y los argumentos recibidos, la cual contacta al servidor para obtener el resultado de la operación.
- `_invoke(method, params)`: Invoca el método remoto con sus parámetros correspondientes y al servidor indicado. Incluye:
  - Construye el cuerpo xml del método, según la especificación XML-RPC, con su nombre y parámetros correspondientes. Usa la función `construir_llamado_xml(method, params)` de la librería **XMLRPC_redes**.
  - Construye el llamado HTTP adjuntando el cuerpo xml anterior. Usa la función `construir_llamado_http(xml)` de la librería **XMLRPC_redes**.
  - Envía los datos a servidor y recibe su respuesta.
  - Parsea la respuesta http y luego el cuerpo xml usando las funciones correspondientes para cada una de estas acciones de la librería **XMLRPC_redes**.
  - Retorna error indicado o el resultado recibido, según corresponda.
- `connect(address, port, timeout)`: función pública para inicializar el cliente. Recibe la dirección ip, el numero de puerto y un timeout en segundos, este ultimo parámetro se puede omitir y se le asignaran 10 segundos de timeout al cliente por defecto.

6.7.4. Cliente

La clase `Client` permite conectarse a un servidor remoto y realizar llamadas a procedimientos.

- `connect(address, port)`: establece conexión con el servidor y devuelve un objeto conexión.
- Uso de `__getattr__` y `__call__` en Python para permitir invocar procedimientos dinámicamente.

Ejemplo de uso:

```
conn = connect('127.0.0.1', 8080)
resultado = conn.sumar(2, 3)
```

6.7.5. Servidor

La clase `Server` permite publicar procedimientos y atender solicitudes.

- `Server((address, port))`: crea un servidor.
- `add_method(proc)`: agrega un procedimiento disponible.
- `serve()`: inicia el servidor en modo bloqueante.

Ejemplo de uso:

```
def sumar(a, b): return a + b

server = Server(('127.0.0.1', 8080))
server.add_method(sumar)
server.serve()
```

6.8. Decisiones de diseño e implementación

6.8.1. Manejo de hilos

Desde el servidor se decidió usar un hilo por cada conexión entrante para soportar múltiples clientes concurrentes. El bucle principal del servidor queda a la escucha en el socket.

6.8.2. Sockets y transporte

Se usaron sockets TCP de la API POSIX.

- El cliente abre un socket al puerto configurado.
- El servidor escucha en el puerto especificado y acepta conexiones entrantes.

6.8.3. HTTP

- Se implementó un parser HTTP mínimo para aceptar únicamente solicitudes POST tal como especificado en el protocolo **XML-RPC**.
- Se usaron los encabezados esenciales: `Host`, `Content-Type`, `Content-Length`.
- Se requiere que estos encabezados existan y sean correctos, segun se especifica en el protocolo **XML-RPC**.
- Se omitieron cabeceras no requeridas para simplificar.
- Las respuestas siempre se crean con la linea '`200 OK`' ya que decidimos que el manejo de errores se realice unicamente mediante el cuerpo xml.

6.8.4. Manejo de errores

Se implementaron los códigos de errores como se especifica en la letra del obligatorio:

1. Error de parseo de XML.
2. No existe el método invocado.
3. Error en parámetros del método.
4. Error interno en la ejecución del método.
5. Otros errores.

Cada error genera una respuesta en formato XML-RPC estándar según el protocolo, con su debido `faultCode` y `faultString`.

6.9. Tipos soportados y serialización (marshalling)

- Se usaron librerías de Python para parsear XML de forma segura.
- Se soportan todos los tipos especificados por el protocolo XML-RPC:
  - Integer
  - Boolean
  - String
  - Float
  - Date/Time
- También se soportan arreglos y enumerados con elementos de estos tipos.
- La serialización sigue estrictamente el formato de la especificación.

6.10. Pruebas realizadas

- Separamos los test en tres archivos:
  - `myServer.py`, `myServer2.py`: crean y levantan los servidores, creando sus respectivos métodos y asignándoles una socket par luego escuchar en él.
  - `myClient.py`: crea un cliente y establece una conexión con cada servidor creado anteriormente, luego ejecuta distintos casos de prueba sobre ambos, cubriendo tanto casos exitosos como casos de errores esperados
- Llamadas correctas a procedimientos remotos.
- Invocación de métodos inexistentes (genera `FaultCode 2`).
- Parámetros inválidos en la invocación (genera `FaultCode 3`).
- Capturas con `tcpdump` y análisis en `WireShark` verificando el formato XML-RPC.

6.11. Conclusiones de la Parte 2

La biblioteca cumple con los requisitos del protocolo XML-RPC y se diseñó de forma modular para permitir ampliaciones futuras. Se identificaron posibles mejoras: soporte a más tipos de datos XML, manejo de cabeceras HTTP adicionales y optimización de concurrencia.

7. Parte 3

7.1. Introducción

En esta parte se buscó integrar las primeras dos partes de la tarea, es decir, se busco probar la biblioteca implementada en la parte 2, utilizando los conceptos estudiados en la parte 1.
Para ello se probo la biblioteca en un entorno realista, levantando dos servidores, un cliente que realiza operaciones a ambos servidores, y 5 hosts que actúan como routers intermediarios entre el cliente y los servidores.

7.2. Decisiones de diseño e implementación

Topología utilizada: como fue propuesto en la letra, se utilizo una topología de dos servidores, un cliente, y 5 routers intermedios
Direcciones de los servidores: se levantaron los dos servidores en las direcciones 150.150.0.2 y 100.100.0.0 respectivamente
Archivos de prueba: Se crearon un archivo para cada servidor "myServer.pyz "myServer2.py", y un archivo para el cliente "myClient.py", que es quien realiza las pruebas a los diferentes servidores
Carácter de las pruebas: se busco lograr generalidad en el diseño de las pruebas, generando pruebas en ambos servidores, de distintos caracteres, como ser pruebas con operaciones correctas, pruebas con parámetros incorrectos, pruebas de concurrencia, entre otras (se detallaran mas adelante)
Capturas de red: se realizaron capturas durante toda la interacción, capturando el trafico tanto en el servidor 1, como en los 5 routers intermedios.

7.3. Reporte de Pruebas

Las pruebas fueron realizadas en el archivo 'myClient.py', que emula un cliente que hace solicitudes a los dos servidores de pruebas. Estas pruebas buscaron brindar robustez al diseño de los servidores.

7.4. Pruebas de éxito al servidor 1 'test_basic_calls'

Estos test fueron simples: una suma, una resta y un concat

7.5. Pruebas de error al servidor 1: 'test_error_cases'

Al mismo servidor, se lo testeo con diferentes casos de error para chequear que efectivamente se devuelvan errores con su faultcode
Los casos de prueba de error consistieron en:

- Llamada a una función inexistente
- Operación con menos parámetros de los requeridos (suma vacía)
- Operación con mas parámetros de los requeridos (suma con 3 números)
- Operación con parámetros inválidos (suma con letras)

7.6. Pruebas de robustez: 'test_new_methods'

Se diseñaron nuevos casos de prueba para seguir la rúbrica proporcionada por el equipo docente, combinada con algunos casos de prueba extras
Estas pruebas consistieron en:

a) Casos de prueba de éxito

- Método sin parámetros: get_current_year(), devuelve el año actual
- Método de un parámetro entero y un string que devuelve un único valor: repeat_string(int, string), repite el string la cantidad de veces del int pasado por parámetro
- Método echo: echo_large_text(large_text), este método recibe un string y lo devuelve. Se utilizo la librería de python lorem_text para generar un texto de 20.000 palabras, que el cliente envía al servidor, y luego recibe la respuesta, se verifica el éxito midiendo el largo del texto recibido, y comprobar si coincide con el largo del texto enviado
- Método que tarde mas de 10 segundos: slow_method(), se usa la función time.sleep(int) para hacer que demore

b) Casos de error

- División entre cero
- Invocación a un método inexistente (realizado en test_error_cases)
- Invocación a un método con parámetros incorrectos. (realizado en test_error_cases)

7.7. Prueba 1 de concurrencia: 'test_concurrency'

En este test de concurrencia se crearon 5 hilos en simultáneo que representaron clientes concurrentes, el objetivo de esta prueba es verificar el correcto manejo de diferentes hilos por parte del servidor. El test consistió en crear 5 hilos, los cuales cada uno de ellos llame al método 'slow_method(5)', es decir, que esperen 5 segundos de manera concurrente en el mismo servidor

7.8. Prueba 2 de concurrencia: 'test_concurrency_mixed_operations'

Este test busco extender la prueba de concurrencia anterior, incorporando diferentes elementos de las pruebas anteriores.
Consistió en crear 5 hilos independientes, y que cada uno realice una operación diferente, buscando que tres de ellos realicen operaciones correctas, y dos de ellos realicen operaciones incorrectas, y verificar que las salidas sean las esperadas.

7.9. Pruebas en el Servidor 2: test_server2_methods

Estas pruebas fueron para verificar que el cliente puede conectarse y realizar operaciones al servidor 2, que se encuentra en la ip '100.100.0.2'. Las pruebas realizadas fueron similares a las pruebas básicas del server 1, consistiendo en:

- Llamadas válidas:
  - power(2, 8)
  - join_with("", [1,2,3])
  - to_upper(rpc", 3)
- Llamadas inválidas:
  - join_with(", "NO_es_lista")
  - conn_b.to_upper("hola", "NO_es_int")
  - conn_b.power('a', 'b')

Las siguientes pruebas buscaron generar diferentes códigos de errores que no fueron contemplados en las pruebas anteriores.
Como es sabido, la clase 'client' siempre genera tanto headers HTTP correctos, como XML correctos en el cuerpo de la request, por tanto es, que decidimos diseñar estas pruebas, a fin de forzar incoherencias, tanto en los headers HTTP, como los llamados XML.
Cabe destacar que en todos estos nuevos tests, se abrió un nuevo socket, con una nueva request HTTP (correcta o incorrecta según el caso), y con un mensaje XML (correcto o incorrecto según corresponda)

7.10. Pruebas llamado XML invalido: 'test_invalid_xml'

En este caso se armó una request HTTP con todos sus headers correctos, pero en el cuerpo se introdujo un mensaje XML con errores (optamos por una etiqueta incorrecta)

7.11. Pruebas request HTTP invalido: 'test_invalid_http_get' y 'test_invalid_http_bad_headers'

Estas pruebas, como indican sus nombres, sirvieron para probar los casos de error de diferentes maneras para las requests HTTP. El primer test mencionado, mandó una request correcta pero usando 'GET' en lugar de 'POST', y el segundo, simplemente modificó los headers con palabras aleatorias .

7.12. Prueba de ejecucion de varios clientes concurrentes

Como último test, se lanzaron varios clientes en simultaneo, usando el cliente 'client_demo.py', que ejecuta 'slow_method ' con 10 segundos de retardo (para así dar tiempo de levantar mas clientes). Se lanzaron con los siguientes comandos:

```
client python3.8 ./client_demo.py 1 &
client python3.8 ./client_demo.py 2 &
client python3.8 ./client_demo.py 3 &
client python3.8 ./client_demo.py 4 &
client python3.8 ./client_demo.py 5
```

Se optó por dejar el último cliente en primer plano, para así observar como cada uno de los clientes lanzados imprimía el mensaje de que terminó la ejecucion del slow_method.

7.13. Resultados observados y Conclusiones

La biblioteca implementada en la parte 2 funciona correctamente en un entorno de red realista con varios servidores y varios clientes de forma concurrente.
Luego de ejecutar el cliente (myClient.py) en primer plano, para poder observar las salidas de los casos de prueba (ver anexo con capturas de pantalla). En donde pudimos ver que todos los casos propuestos fueron realizados de forma exitosa.
Se probó la capacidad de concurrencia del servidor 1, pudiendo soportar 5 hilos del cliente realizando diferentes operaciones. También destacar que se devolvieron los faultcodes correspondientes para cada caso de prueba de error:

- Error de parseo XML (XML inválido): faultCode 1
- Error de método inexistente: faultCode 2
- Error en los parámetros usados en el llamado: faultCode 3
- Error Interno (en la ejecución de la función): faultCode 4
- Otro error (por ejemplo headers HTTP inválidos): faultCode 5

Se lograron, también de forma exitosa, las pruebas de clientes concurrentes en el servidor 1.

8. Conclusiones Generales

A grandes rasgos, concluimos que fueron logrados todos los objetivos. Conseguimos entender y aprender a utilizar todas las herramientas de captura y análisis de paquetes, tanto como las herramientas de análisis de conexión. Logramos implementar en su totalidad la biblioteca xmlrpc_redes y su utilización de la misma para generar servidores y clientes mediante una red emulada en mininet la cual soporta múltiples clientes y servidores, así como nodos intermedios por los cuales los paquetes viajan al realizar las conexiones cliente-servidor mediante el protocolo utilizado, http.

9. Referencias

Usa formato estándar (APA, IEEE, etc.):

- Tanenbaum, A. S. (2011). *Redes de computadoras.*
- *Redes de computadoras: Un Enfoque Descendente 5ta Edición, James F. Kurose*
- `https://www.wireshark.org/`

10. Anexo, capturas de pantalla relevantes

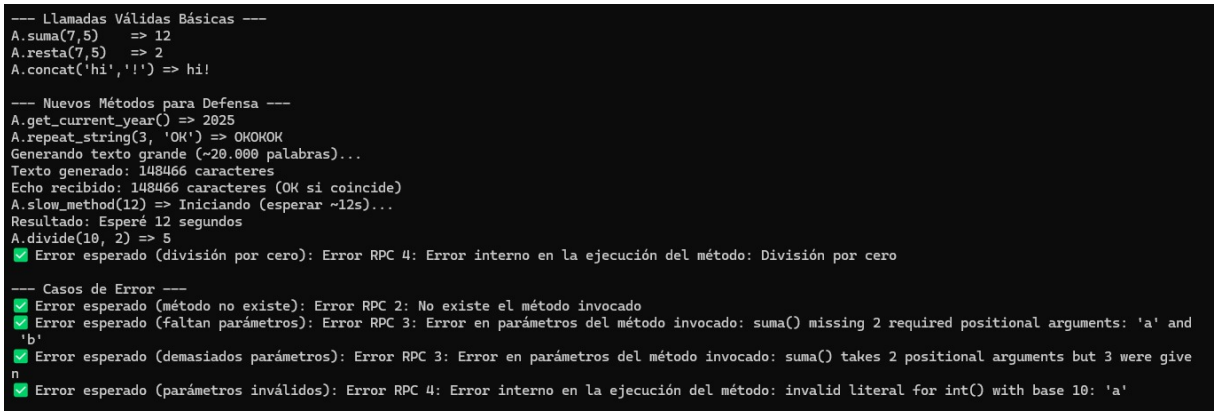

Figura 3: Pruebas básicas de los métodos

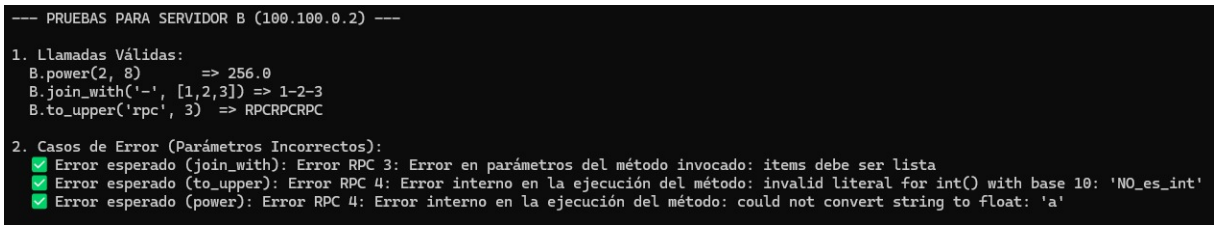

Figura 4: Pruebas servidor B

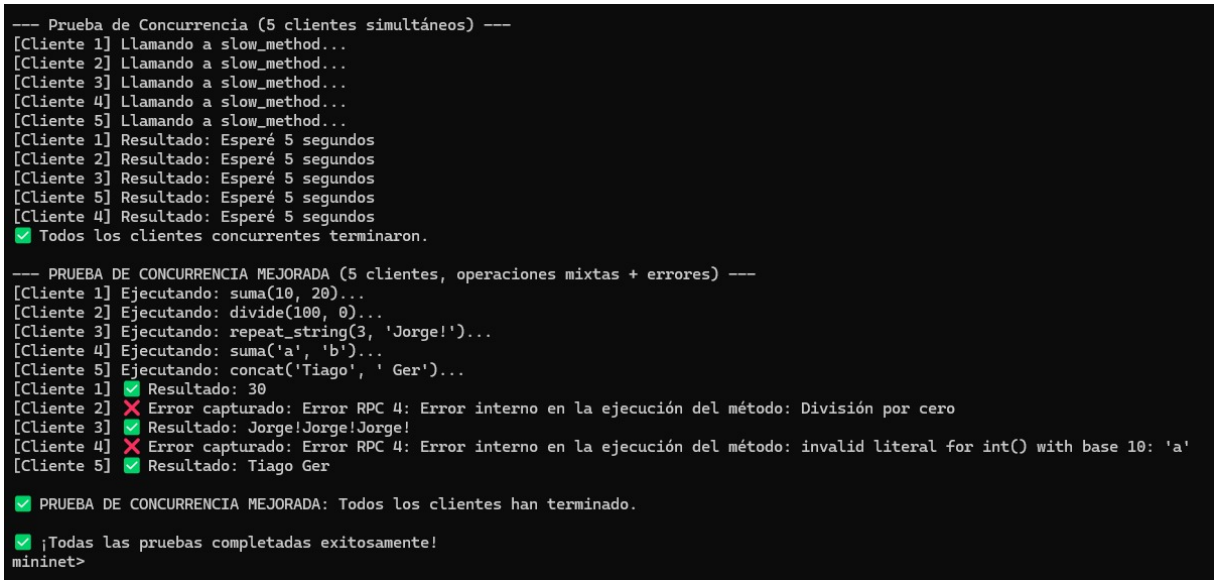

Figura 5: Pruebas de concurrencia