Redes de Datos
Laboratorio 1 – Capa de Aplicación

Universidad ORT Uruguay

Curso 2025

En este laboratorio, analizaremos cómo realizar la comunicación entre *procesos* a nivel de capa de Aplicación, e implementaremos algunos de los protocolos vistos en el curso.

Para ello, utilizaremos la API que ofrece la Capa de Transporte, es decir, la posibilidad de establecer la comunicación entre un proceso cliente y el proceso servidor, para luego enviar datos sobre el mismo.

Utilizaremos:

- Scripts en lenguaje `python` para implementar la comunicación.
- El software Wireshark para capturar y analizar el tráfico de la red.
- Un *servidor*, implementado como una máquina virtual (Virtual Machine, VM) que corre en el sistema host, y que tiene instalado todos los servicios.

Preparación del ambiente

VM Servidor

La VM se ejecuta usando el software de virtualización VirtualBox. Se proporciona un archivo de tipo `.ova` que contiene la VM. Este servidor consiste en un Linux (Fedora 42 Server) con acceso por consola. El usuario administrador es `redes` con password `redes`, y dispone de otros usuarios de la forma `ort-grupo1`...`ort-grupo4`, con password idéntico.

1. Inicie VirtualBox e importe el servicio virtualizado `Servidor Redes 2025` del archivo `.ova`.
2. Inicie la VM en VirtualBox.

Una vez pronto el ambiente, en su equipo va a contar con un nuevo adaptador de red el cual servirá únicamente para la comunicación entre la máquina host y la VM. Dicha interfaz, de tipo VirtualBox Host Only Network proporciona una interfaz virtual de red con las siguientes direcciones:

- IP Host: `192.168.56.1`,
- IP VM (servidor): `192.168.56.2`.

Para visualizar el tráfico entre la máquina host y el servidor, podrá capturar paquetes utilizando Wireshark en esta interfaz virtual. A su vez, la VM cuenta también con otra interfaz de red a través de la cual se conecta a Internet.

Scripts cliente

Descargue en la máquina host los scripts de `python` de ejemplo que se proporcionan en la página del curso, y colóquelos en una carpeta de trabajo. Para ejecutar estos scripts puede utilizar:

Windows: `py <script> <parámetros>`

Mac/Linux: `python <script> <parámetros>`

Parte 1: Comunicación entre procesos

El objetivo de esta parte es ilustrar una comunicación básica entre procesos utilizando *sockets*. El socket provee una abstracción de las capas inferiores permitiendo la comunicación. En nuestro caso utilizaremos sockets *orientados a conexión* mediante el protocolo TCP.

Los pasos básicos para crear un socket *cliente* son:

```
from socket import * #importa la biblioteca
clientSocket = socket(AF_INET, SOCK_STREAM) #crea un objeto socket TCP
clientSocket.connect((serverHost,serverPort)) #inicia la conexión
```

Aquí las variables `serverHost` y `serverPort` son la dirección IP y el puerto donde está el servidor escuchando.

Una vez creado el socket podemos utilizar:

```
clientSocket.sendall(datos) #envía un stream de bytes
receivedBytes = clientSocket.recv(N) #recibe hasta N bytes del socket
```

Observación: `clientSocket.recv(N)` bloquea la ejecución hasta recibir datos. Por ello, puede ser importante fijar un valor de *timeout* de manera de que el programa no quede bloqueado esperando datos para siempre. Esto puede hacerse con `clientSocket.settimeout(t)` para limitarlo a `t` segundos.

1. Analice el script de ejemplo `tcp_socket_client.py` para entender su funcionamiento.
2. Ingrese en la VM como administrador (`redes`), y ejecute el script de servidor:

   ```
   $> python socket_examples/threaded_tcp_socket_server.py <puerto>
   ```

   El valor de puerto es donde el proceso queda escuchando el proceso. Los valores habilitados son 5000 a 6000.
3. En la máquina host, comience a capturar paquetes en Wireshark, en la interfaz de Virtualbox host-only network.
4. En una consola, ejecute el script `tcp_socket_client.py` indicando la IP y puerto donde debe conectarse.
5. Intercambie mensajes con el servidor y vea lo que ocurre en la captura de paquetes. Identifique los mensajes de ida y retorno. Analice la composición de los mismos. ¿Qué hace el servidor con los mensajes enviados?
6. Finalice con `Ctrl+C` el script y vea qué ocurre en la captura.
7. Inicie ahora en dos consolas diferentes el script `tcp_socket_client.py`, conectándose al mismo servidor y en el mismo puerto. ¿Logra que ambas funcionen simultáneamente?
8. ¿Cómo hace el servidor para distinguir una conexión de la otra?

Parte 2: HTTP

En esta parte intercambiaremos ahora mensajes con un servidor HTTP, mediante el protocolo visto en clase. El servidor está en la VM ya configurado y escucha en el puerto estándar de HTTP, 80. Capture el tráfico para ver los intercambios.

1. Tomando como base el script `tcp_socket_client.py`, cree un script `http_client.py` que realice un pedido GET de HTTP/1.0 al servidor y reciba e imprima la respuesta. ¿Logra obtener la página del laboratorio completa?
2. Modifique el script anterior para realizar el pedido GET por HTTP/1.1. No olvide el parámetro obligatorio `Host:<nombre>` de HTTP/1.1. Vuelva a realizar el pedido. ¿Logra respuesta? ¿Qué diferencias observa?

3. Vuelva a realizar el pedido de la página índice del servidor utilizando un navegador (no olvide quitar el proxy HTTP si éste está configurado). Explique las diferencias observadas. ¿Cuántos pedidos realiza el navegador? ¿Cuántos son exitosos?

4. Utilizando la biblioteca de `python urllib` (ver Apéndice), modifique el script anterior para que reciba una URL como parámetro y realice el GET correspondiente via HTTP/1.1. Pruebe obtener una página remota.

Parte 3: Envío de correo

El protocolo SMTP de envío de correo visto en clase se diferencia de HTTP en que establece un *diálogo* entre cliente y servidor para lograr el envío. Un ejemplo de diálogo simple SMTP puede verse en el material del curso.

En la VM está configurado un servidor SMTP (Postfix) escuchando en el puerto estándar 25. Este servidor puede enviar correos entre las direcciones `ort-grupox@lab.ort.edu.uy`.

1. Tomando como base el script `tcp_socket_client.py`, implemente un script que:

   a) Consulte al usuario los campos de quién envía el correo, destinatario y subject, y luego el mensaje a enviar (acotado a una línea).

   b) Establezca el socket apropiado para conectar con el servidor SMTP.

   c) Realice el diálogo SMTP enviando y recibiendo los datos según el protocolo. No olvide incluir los campos de encabezado y mensaje dentro de los datos del correo a enviar.

   d) Cierre la conexión SMTP y luego el socket de manera apropiada.

2. Pruebe utilizar el script anterior para enviar un correo entre dos usuarios utilizando el servidor SMTP de la VM.

3. ¿El servidor requiere autenticación? ¿Qué puede decir sobre la seguridad de este protocolo?

Parte 4: Lectura de correo

Para leer el correo de la parte anterior, se proporciona el script `simple_pop3_client.py` que implementa un cliente básico POP3, el protocolo más simple de lectura de correos. El servidor tiene instalado un servidor POP3 (Dovecot) escuchando en el puerto estándar 110.

1. Analice el script `simple_pop3_client.py` para comprender su funcionamiento.

2. Intente recuperar el o los mensajes enviados en la parte anterior utilizando POP3.

3. Capture los paquetes para ver el intercambio realizado. En particular observe que el password va en texto plano por la red.

Parte 5: SSH

En esta parte, exploraremos un protocolo seguro de consola remota: SSH (Secure SHell). Este protocolo encripta el tráfico entre cliente y servidor y escucha en el puerto 22.

1. Capture el tráfico en la interfaz con la VM

2. Realice en la máquina host una conexión de consola remota mediante SSH utilizando un cliente adecuado, por ejemplo:

   Windows `putty.exe`.

   Mac, Linux Ejecutar `ssh redes@192.168.56.2` en la consola.

3. Observe el tráfico capturado y vea cómo comienza a encriptarse al avanzar la comunicación.

Apéndice: Comandos útiles

Conexión mediante sockets

Los siguientes métodos son útiles para trabajar con sockets:

```
from socket import *

clientSocket = socket(AF_INET, SOCK_STREAM) #crea un socket TCP
clientSocket.settimeout(10.0) #setea el timeout en 10s
clientSocket.connect((host,port)) #establece la conexión a host/puerto.
clientSocket.sendall(<bytes>) #envía una cadena de bytes
clientSocket.recv(N) #recibe hasta N bytes del socket.
clientSocket.close() #cierra la conexión.
```

Nota: `socket.recv(N)` retorna vacío cuando el otro extremo de la conexión se desconecta.

Cadenas de bytes

En `python`, los sockets requieren cadenas de bytes para enviar/recibir. Pueden ser útiles los siguientes métodos:

```
    data = b''  #crea una cadena de bytes vacía
    data = data+nuevosDatos #agrega nuevos datos a una cadena de bytes

    ##Conversión de datos

    #Crear una cadena de bytes que codifica el mensaje
    #ingresado como string.
    data = string.encode()

    #Decodificar una cadena de bytes en un string adecuado:
    string = data.decode()
```

Manejo de strings

```
    #Crear un string
    string = "Hola mundo"

    #Concatenar strings
    string = "Hola"+" "+"mundo"

    #Caracter de retorno de linea
    crlf = "\r\n"

    #Pasar variables a strings
    string = f"El valor de x es {x}"
```

Lectura de URLs

```
    from urllib.parse import urlparse

    url = urlparse(<URL>)
```

```
    url.hostname #hostname de la URL
    url.port #puerto de la URL, en caso de estar especificado
    url.path #ubicación del elemento buscado dentro del hostname
```