Introducción Prácticas IRD

1. Introducción

Este documento es una introducción a las prácticas de Internet Redes y Datos. Contiene en un primer lugar una exposición de las herramientas necesarias para el desarrollo de las prácticas de programación (netcat y git) así como el enunciado del ejemplo de un sistema cliente servidor de eco para UDP.

En este documento se proporciona la información necesaria para hacer uso de las dos herramientas que ayudarán a la realización de las prácticas de programación. Además se propone una organización para el desarrollo y el ciclo de trabajo de las prácticas.

2. Netcat

Netcat o nc es una herramienta que permite leer y escribir datos a través de conexiones de red, haciendo uso del protocolo TCP/IP.

Para su instalación en Windows se deberá descargar MobaXterm del siguiente enlace, no es necesario instalarlo, netcat está disponible si se clica en "Start local terminal":

https://download.mobatek.net/1242019111120613/MobaXterm_Portable_v12.4.zip

En Linux suele estar presente en la línea de comandos, en caso contrario debería encontrarse en el repositorio del sistema operativo utilizado.

3. Git

Git es el software de control de versiones a emplear para el desarrollo de las prácticas, así como el repositorio dónde se han de realizar las entregas. En este caso, usaremos como repositorio el proporcionado por https://github.com.

Para su instalación se ha de descargar (https://git-scm.com/downloads) e instalar utilizando las opciones por defecto. Cabe destacar que en determinados entornos es posible instalarlo directamente desde los repositorios de software de la distribución. Por ejemplo, en Ubuntu:

```
sudo apt-get install git
```

Configuración básica (en Windows, los siguientes comandos deben ejecutarse desde *git-bash* en el directorio raíz de Git):

```
git config --global user.email "tu_correo_electronico@udc.es"
git config --global user.name "Tu Nombre"
```

Es posible configurar el editor de texto por defecto para Git, por ejemplo para configurar Sublime como editor por defecto (puede descargarse en: https://www.sublimetext.com/3)

```
> Windows:
git config --global core.editor "'C:\Program Files\
                    Sublime Text 3\sublime_text.exe' -w"

> Linux:
git config --global core.editor "subl -w"
```

3.1. Creación y configuración de claves SSH

- Desde el interprete de *git-bash* en Windows > Generar claves SSH en la ruta por defecto ($HOME/.ssh) y con nombres por defecto.

  ```
  ssh-keygen -t rsa -b 4096 -C "tu_correo_electronico@udc.es"
  ```

- Desde un navegador, accediendo a: https://github.com/settings/keys/new
- En el campo "Key" se debe copiar la clave pública (i.e. el contenido del fichero *$HOME/.ssh/id_rsa.pub*)
- En el campo "Title" se podrá especificar un nombre para la clave.
- Clic en el botón "Add SSH key"

Por último se ha de comprobar la conectividad contra el servidor Git y añadirlo a la lista de "known-hosts".

```
ssh -T git@github.com
```

3.2. Creación de repositorio

1. Desde un navegador, acceder a:
   https://github.com/users/<user-login>/
   substituyendo <user-login> por el usuario.
2. Se ha de hacer clic en "+" (en el lado derecho de la caja de búsqueda, en la parte superior de la página)
   a) Clic en "New repository"
   b) Se deberá indicar "ird-practicas" como nombre del repositorio.
   c) El tipo de proyecto debe ser **Privado**.
   d) Una vez creado el proyecto acceder al menú "Settings".
   e) En el menú izquierdo acceder a "Manage access".
   f) Invitar como colaborador a **"manuel.fernandezl@udc.es"**.

3.3. Inicializar el repositorio

Se deberá crear un directorio llamado "ird-practicas" y una vez dentro del mismo, ejecutar los siguientes comandos:

```
git init
git remote add origin git@github.com:<user-login>/
            ird-practicas.git
git add .
git commit
git push -u origin master
```

Hay que recordar que <user-login> debe substituirse por el usuario.

4. Estructura prácticas de programación

Para el desarrollo de las tres prácticas de programación en Python se deberá seguir la siguiente estructura de directorios:

```
./ird-practicas
./ird-practicas/tutorial
./ird-practicas/socketsTCP
./ird-practicas/servidorWeb
./ird-practicas/serviciosWeb
```

Generando un fichero para cada ejercicio propuesto en el tutorial y en la práctica de Sockets TCP y un único fichero para el desarrollo del servidorWeb y el servicioWeb, donde se implementarán las sucesivas iteraciones.

5. Guía uso Git

- *git status* → Muestra el estado de los ficheros
- *git add <recurso>* → Añade el recurso indicado al índice
- *git commit* → Registra los cambios en el repositorio. Se puede emplear la opción *-m* para indicar el mensaje, de la siguiente manera:

```
git commit -m "Mensaje de commit."
```

- *git push* → Actualiza los cambios realizados en el repositorio remoto.

6. Introducción Sockets UDP

6.1. Conceptos básicos

6.1.1. Pila de protocolos TCP/IP

La pila de protocolos TCP/IP, que permite la transmisión de datos entre redes de computadores consta de una serie de capas como se puede apreciar en la siguiente tabla:

Nivel	Protocolos
Aplicación	HTTP, FTP, SSH, Telnet
Transporte	TCP, UDP
Red	IP, ICMP, ARP
Acceso al medio	Ethernet

De manera habitual, cuando se escriben aplicaciones en red, se trabajará a nivel de aplicación y se utilizarán además protocolos del nivel de transporte. Por ello es importante conocer las principales diferencias entre estos.

TCP (Transmission Control Protocol):

- Protocolo orientado a conexión.
- Provee un flujo de bytes fiable.
- Protocolos a nivel de aplicación: telnet, HTTP, FTP, SMTP, ...

UDP (User Datagram Protocol):

- Protocolo no orientado a conexión.
- Envía paquetes de datos (datagramas) independientes y sin garantías.
- Permite broadcast y multicast.
- Protocolos a nivel de aplicación: DNS, TFTP, ...

6.1.2. Sockets

Cuando trabajamos en una red de ordenadores y queremos establecer una comunicación (enviar o recibir datos) entre dos procesos que se están ejecutando en máquinas diferentes de dicha red necesitamos hacer uso de los *Sockets* para abstraer las conexiones.

Para diferenciar estas conexiones se les asigna un número de 16 *bits* (entre 0 y 65535) conocido como puerto. Por tanto para dirigir de manera unívoca la información entre dos procesos en diferentes máquinas hemos de hacer uso de la dirección IP y el puerto asignado en cada máquina, así como del protocolo empleado.

Según el papel que cumplan los programas podremos hablar de cliente o servidor, siendo el primero el que inicia una petición y recibe la respuesta y el

segundo el que se encuentra a la espera de recibir una conexión para generar una respuesta.

Se puede consultar la documentación relativa a Sockets en Python en el siguiente enlace:

https://docs.python.org/3/library/socket.html

6.2. Comandos

Resulta útil hacer uso de un comando para la prueba de los programas que vamos a desarrollar en las prácticas, es el *nc* o *Netcat*. Este programa nos permite enviar datos a través de una red, indicando el destino y el protocolo a emplear, entre otras opciones.

Especialmente útiles pueden resultar las siguientes combinaciones:

```
# Abrir un servidor UDP escuchando en un determinado puerto
> nc -u -l -p <puerto>
# Abrir un cliente UDP para enviar un mensaje
> nc -u <ip> <puerto>
```

7. Ejercicios

7.1. Sockets UDP

Los *sockets* UDP son no orientados a conexión, no pudiendo garantizarse la recepción de los paquetes (datagramas) ni el orden de los mismos. Podemos definir por tanto un **datagrama** como un mensaje independiente, enviado a través de la red cuya llegada, tiempo de llegada y contenido no están garantizados.

7.1.1. Cliente UDP

Crearemos un programa que reciba como argumentos la máquina destino, el puerto y el mensaje a enviar, los envíe y muestre el mensaje de respuesta recibido. Para ello proponemos el siguiente código:

Primero importaremos las librerías necesarias:

```
import sys
import socket
```

A continuación definiremos la función *main()* que incluirá todo el código que queramos ejecutar y que comenzará comprobando si el programa recibe el número adecuado de argumentos.

```
def main():
    if len(sys.argv) != 4:
        print("Formato ClienteUDP <maquina> <puerto> <mensaje>")
        sys.exit()
```

Dentro de esa función se añadirán las siguientes instrucciones haciendo uso de la estructura *try-except-finally*.

```
try:
    # Instrucciones sockets
    ...
except socket.timeout:
    # Captura excepción si el tiempo de espera se agota.
    print("{} segundos sin recibir nada.".format(timeout))
except:
    # Captura excepción genérica.
    print("Error: {}".format(sys.exc_info()[0]))
    raise
finally:
    # En cualquier caso cierra el socket.
    socketCliente.close()
```

A continuación veremos las instrucciones necesarias para crear el socket UDP, enviar un mensaje, recibir la respuesta e imprimir el contenido de la misma.

Primero obtenemos los argumentos necesarios para crear el socket (maquina y puerto) y el mensaje a enviar.

```
# Leemos los argumentos necesarios
maquina = sys.argv[1]
puerto = int(sys.argv[2])
mensaje = sys.argv[3]
```

A continuación y utilizando los valores que hemos leído creamos un socket no orientado a conexión, esto es, un socket UDP, mediante la función *socket.socket()*, pasando como argumento el tipo de socket que queremos crear *socket.AF_INET,socket.SOCK_DGRAM*.

```
# Creamos el socket no orientado a conexión
socketCliente = socket.socket(socket.AF_INET,
                              socket.SOCK_DGRAM)
```

Asignamos al socket un timeout de 300 segundos.

```
# Establecemos un timeout de 300 segs
timeout = 300
socketCliente.settimeout(timeout)
```

Mostramos el mensaje y a dónde se enviará:

```
print("CLIENTE: Enviando {} a {}:{}".format(
    mensaje,maquina,puerto))
```

Enviaremos el mensaje mediante la función *sendto()*, codificando el contenido en *'UTF-8'* para evitar problemas de decodificación en el reenvío y recepción del mensaje.

```
# Enviamos el mensaje a la máquina y puerto indicados
socketCliente.sendto(mensaje.encode('UTF-8'),
                        (maquina, puerto))
```

A continuación leeremos el mensaje de respuesta recibido en el cliente mediante la función *(*recvfrom()), indicando la longitud del mensaje a recibir (se puede calcular con la función *(*len())).

```
# Recibimos el mensaje de respuesta
mensajeEco, a = socketCliente.recvfrom(len(mensaje))
```

Una vez leída la respuesta, imprimiremos el contenido así como la máquina y puerto que la envía, decodificando el mensaje con la misma codificación utilizada en el paso de envío del mensaje.

```
print("CLIENTE: Recibido {} de {}:{}".format(
    mensajeEco.decode('UTF-8'),maquina,puerto))
```

Por último, para que se realice la ejecución de este código y se le pasen los argumentos definidos incluiremos lo siguiente, que detectará si el fichero se está utilizando como un ejecutable y llamará a la función que hemos definido antes.

```
if __name__ == "__main__":
    main()
```

De manera más esquemática se exponen a continuación los pasos a ejecutar en el cliente:

1. Se crea un *socket* no orientado a conexión para el envío y recepción de mensajes, para ello se indica *socket.AF_INET* para indicar que se trata de *IPv4* y *socket.SOCK_DGRAM* para usar UDP.
2. Establecemos un *Timeout* para el *socket*, de 300 segundos.
3. Enviamos el mensaje codificado a la dirección destino, representada por una tupla (máquina, puerto).
4. Recibimos la respuesta en el mismo *socket* abierto.
5. Decodificamos y mostramos el mensaje recibido.
6. Controlamos la excepción por *timeout (socket.timeout)* y cualquier otra excepción posible.
7. En cualquier caso tratamos de cerrar el *socket* al final de la ejecución.

7.1.2. Servidor UDP

En base al cliente de eco UDP implementado en el apartado anterior procederemos a desarrollar un servidor de eco UDP, que recibirá como argumento el puerto en el estará a la escucha.

Al igual que en el caso de la implementación del cliente UDP, empezaremos importando las librerías necesarias.

```
import sys
import socket
```

Definiremos, de manera análoga al punto anterior, la función *main()*, la lectura de argumentos y la estructura *try-except-finally*.

```
def main():

    if len(sys.argv) != 2:
        print("Formato ServidorUDP <puerto>")
        sys.exit()

    try:
        # Instrucciones sockets
        ...
    except socket.timeout:
        print("{} segundos sin recibir nada.".format(timeout))

    except:
        print("Error: {}".format(sys.exc_info()[0]))
        raise

    finally:
        socketServidor.close()
```

Dentro de la estructura *try* comenzaremos por leer el puerto en el cual escuchará el servidor de la siguiente manera:

```
    # Leemos los argumentos necesarios
    puerto = int(sys.argv[1])
```

A continuación creamos el socket de la misma manera en la que lo hicimos para el cliente:

```
    # Creamos el socket no orientado a conexión
    socketServidor = socket.socket(socket.AF_INET,
                                   socket.SOCK_DGRAM)
```

Una vez creado el socket, hay que indicarle en que puerto queremos que escuche (i.e. el que recibimos como argumento del programa), en la máquina en la que nos encontramos.

```
# Asociamos el socket a cualquier dirección local
# en el puerto indicado
socketServidor.bind(("", puerto))
```

De la misma forma que en el cliente, asignamos un valor de timeout al socket:

```
# Establecemos un timeout de 300 segs
timeout = 300
socketServidor.settimeout(timeout)
```

Mostramos por pantalla un mensaje que indique que el servidor ya está escuchando e indicamos en que puerto.

```
print("Iniciando servidor en PUERTO: ",puerto)
```

Para la recepción de mensajes crearemos un **bucle** infinito, para ello declararemos un bucle *while* indicando como condición de terminación del bucle *True*.

Dentro del bucle que acabamos de definir deberemos ejecutar tres acciones: recibir el mensaje, mostrar por pantalla el contenido y reenviar el mensaje recibido de vuelta al emisor del mismo. Para esto utilizaremos las instrucciones mostradas a continuación. En la recepción del mensaje, obtenemos mediante *recvfrom* el mensaje y la dirección de origen. Tal y como indica la documentación relativa a esta función, utilizaremos *2048* como tamaño de buffer recomendado.

```
# Recibimos el mensaje
mensaje, direccion = socketServidor.recvfrom(4096)

print("Recibido mensaje: {} de: {}:{}".format(
    mensaje.decode('UTF-8'),direccion[0],direccion[1]))

# Enviamos el mensaje
socketServidor.sendto(mensaje, direccion)
```

Por último para permitir la correcta ejecución de estas instrucciones así como el paso del argumento necesario, añadiremos al final las siguientes líneas que llamarán a la función *main()*.

```
if __name__ == "__main__":
    main()
```

De manera más esquemática se exponen a continuación los pasos a ejecutar en el servidor:

1. Creamos un *socket* UDP.
2. Vinculamos el *socket* al puerto indicado mediante la función *bind*.
3. Establecemos el *timeout*.

4. Creamos un bucle infinito:
    a) Recibimos el mensaje mediante la función *recvfrom*.
    b) Mostramos el contenido del mensaje.
    c) Enviamos el mismo mensaje a la dirección indicada por el *socket*.
5. Controlamos la excepción por *timeout* y cualquier otra excepción posible.
6. Cerramos el *socket* al final de la ejecución.