



UNIVERSIDAD NACIONAL DE CÓRDOBA
Facultad de Ciencias Exactas, Físicas y Naturales

Cátedra de Sistemas Operativos II

TRABAJO PRÁCTICO 1
Comunicación con Sockets
Mauricio Aguilar

17 de Abril de 2016

Índice

[Introducción](#)

[Descripción del problema](#)

[Requerimientos y tareas](#)

[Diseño y documentación](#)

[Implementación y Resultados](#)

[Conclusiones](#)

Introducción

Cuando hablamos de Socket nos referimos a un concepto por el cual dos programas (posiblemente situados en computadoras distintas) pueden intercambiar cualquier flujo de datos, generalmente de manera fiable y ordenada.

Los sockets de Internet constituyen el mecanismo para la entrega de paquetes de datos provenientes de la tarjeta de red a los procesos o hilos apropiados. Un socket queda definido por un par de direcciones IP local y remota, un protocolo de transporte y un par de números de puerto local y remoto.

La comunicación por sockets sigue el modelo cliente-servidor.

- El cliente debe conocer la dirección del servidor para la comunicación.
- El servidor no conoce la existencia del cliente.
- Una vez establecido el contacto ambas partes pueden enviar y recibir datos, no se reconoce un maestro o esclavo.

Puede darse distintas situaciones como:

- > Un servidor con muchos clientes conectados.
- > Un cliente que se conecta a multiples servidores.

- Socket es una forma de IPC (InterProcess Communication) introducida por la Universidad de Berkeley en su versión de Unix (BSD).
- El concepto de socket nos evita tener que aprender una interfaz de programación diferente para cada protocolo.
- En síntesis, muchos protocolos de comunicación usan esta herramienta.

Descripción del problema

El Servicio Meteorológico Nacional (SMN) ha adquirido una serie de 25 estaciones meteorológicas automáticas (AWS), que serán instaladas por todo el territorio nacional. Cada AWS consta de una serie de sensores (GPS, humedad, temperatura, etc), conectado a un sistema de adquisición de datos (COP), que toma medidas de los sensores (telemetría).

Estos datos se guardan en formato CVS, con el siguiente formato: tiempo, precipitación, humedad relativa, temperatura del aire a 1.5m de altura, temperatura de suelo a 10cm de profundidad

Cada AWS se encuentra conectada a Internet, a los fines de controlarse de forma remota y de poder extraer sus datos. Estas actividades se realizan desde el centro de Operaciones del SMN en Dorrego, CABA.

El SMN solicita a la Escuela de Ingeniería en Computación que diseñe, implemente y testeé el software (desarrollado en C), necesario para realizar la conexión, control y transferencia de datos telemétricos entre el COP y todas las AWS, utilizando una arquitectura cliente servidor.

El programa que corre en el COP debe poder conectarse a una AWS, de forma segura (orientado a conexión), utilizando un puerto fijo (60xx, con xx de 20 a 44 de acuerdo al número de estación), y tomar un número de un puerto libre (aleatorio) de su sistema operativo.

Una vez establecida la sesión entre el COP y la AWS, la aplicación proveerá de un "prompt" que identifica con quién me he conectado. En el contexto de este "prompt" se podrán ejecutar comandos propios de la aplicación.

Los comandos de la aplicación son:

- **ip port:** debe conectarse a la AWS que posea esa ip y a ese puerto
- **disconnect:** finaliza la conexión
- **get_telemetry:** obtiene el último registro de telemetría
- **get_datta:** descarga el archivo (del registro) con todos los datos de telemetría obtenidos hasta el momento
- **erase_datta:** borra la memoria con los datos de telemetría de la AWS

Se pide que la transferencia de archivos se realice con utilizando conexión no segura (sin conexión).

Durante la operación de transferencia del archivo, se debe bloquear la interfaz de usuario del programa, de tal forma que no se puedan ingresar nuevos comandos en la aplicación hasta que no haya finalizado la transferencia. Si no se pudo realizar la conexión, debe avisar en pantalla del error.

Debe incluirse un mecanismo de control y manejo de errores por parte del AWS con comunicación al COP, esto es, si un comando no es reconocido o el argumento de un comando es no válido, debe comunicar esta situación al COP ante la imposibilidad de ejecutarlo. Todos los procesos deben ser mono-thread.

Se debe desarrollar un simulador en la AWS, que vaya entregando cada 1 segundo uno de los registros de telemetría, tomados del archivo adjunto, y guardarlo en un registro (memoria interna), del sistema de adquisición de datos.

A fines de llevar a cabo su implementación, el SMN provee de una plataforma de desarrollo, INTEL Galileo V1, sobre la cual desarrollar el prototipo de ingeniería. A su vez también hace entrega de un archivo con los datos de una estación meteorológica (telemetría), con el cual realizar las pruebas.

El prototipo desarrollado debe permitir la conexión desde el COP con al menos un AWS. Tanto la especificación del protocolo de red en la capa de aplicación, así como la elección de la interfaz de usuario del programa cliente, quedan liberadas a criterio del alumno.

Se deberá proveer los archivos fuente, así como cualquier otro archivo asociado a la compilación, si existiera (archivos de proyecto "Makefile", por ejemplo).

También se debe entregar un informe. Se debe asumir que las pruebas de compilación se realizarán en un equipo que cuenta con las herramientas típicas de consola para el desarrollo de programas (Ej: gcc, make), y NO se cuenta con herramientas "GUI" para la compilación de los mismos (Ej: eclipse).

Requerimientos y tareas

Requerimientos Funcionales:

- Diseñar una aplicación para la AWS que funcione como servidor de datos.
 - Al ejecutar el servidor, se le deberá proporcionar un número de puerto entre 6020 y 6044, que se corresponden con los servidores AWS del 0 al 24, respectivamente.
 - Dado este número de puerto, el servidor deberá generar una conexión por socket internet para la transmisión de los comandos.
 - El servidor creará un proceso hijo al cual le delegará el socket y que atenderá al cliente
 - El comando **get_telemetry** devolverá la información a los últimos datos sensados, que se corresponde con la última línea del archivo registro.csv
 - El comando **get_datta** se encargará de leer toda la información sensada que esta alojada en el archivo registro.csv, generará un socket internet UDP para transmitir todos los datos de dicho archivo al cliente.
 - El comando **erase_datta** borrará el contenido de registro.csv.
 - El comando **disconnect** desconectará al cliente y finalizará al proceso hijo que lo atiende.

- Diseñar una aplicación para la AWS que **simule** la toma de datos desde los sensores y los aloje en un archivo llamado registro.csv
 - El simulador tomará los datos de los sensores cada 1 segundo, y los alojará en un archivo llamado registro.csv.
- Diseñar una aplicación para el COP, que permita ingresar los comandos para operar sobre el servidor.
 - El Centro de Operaciones generará una conexión TCP y se conectará al servidor AWS, introduciendo su ip y el puerto. Para la transmisión del archivo por el comando get_datta, se creará además un socket UDP.
 - El COP enviará a través de dicha conexión los comandos antes mencionados al servidor, el cual ejecutará sus correspondientes acciones.
- Las tres aplicaciones se ejecutará por consola de comandos.

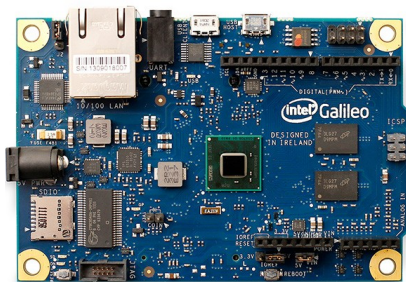
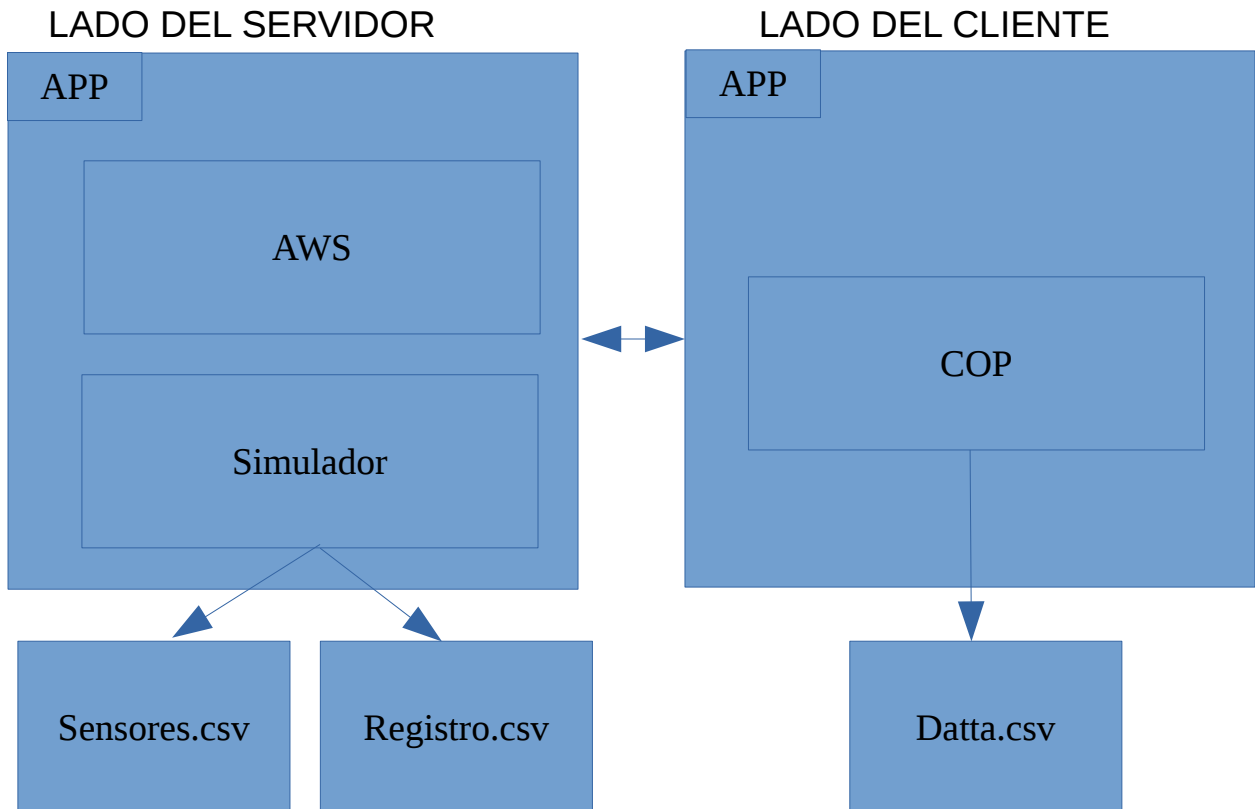
Requerimientos No Funcionales:

Performance: El sistema deberá tener un tiempo de respuesta relativamente bajo, sobre todo en el simulador, que debe recolectar datos de manera periódica y de manera que no se sobrecargue, trabe o pierda datos.

Confiabilidad: El sistema debe mantenerse activo antes diferentes situaciones y errores, como inexistencia de archivos, comandos mal ingresados, puertos erroneos, etc.

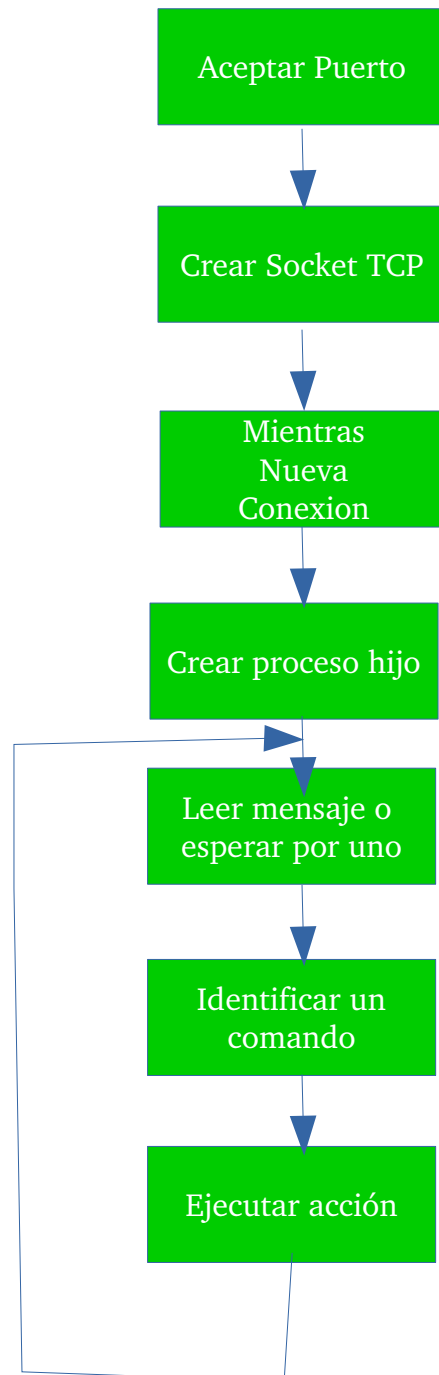
Diseño y documentación

Según lo antes planteado, se ilustra un simple diagrama de componentes para ilustrar la manera en que se relacionan las partes del sistema:

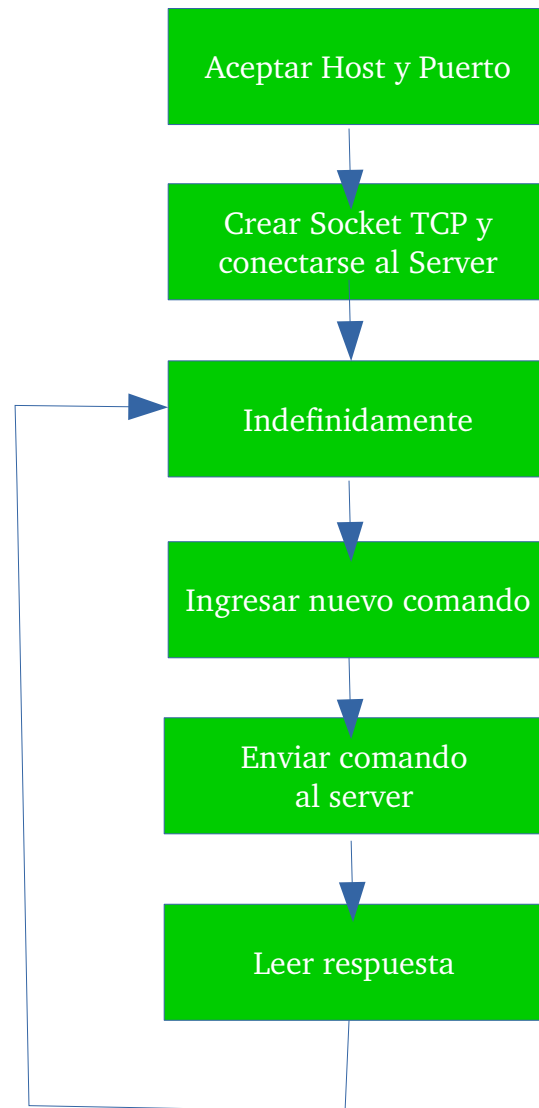


A continuación se presentará un diagrama de flujo de lo que se espera realizar en cada una de las aplicaciones.

AWS.c



COP.c



Se entregan los archivos mencionados en el diagrama de componentes.

Primero procedemos a cargar los archivos en la placa Intel Galileo I:

1 Poner microsd, cable de red, y cable de alimentacion

2 Abrir terminal y conectarse con:

>ssh root@ip

donde ip es la ip de la Galileo, ejemplo: >ssh@192.128.0.104

Ver ip, comando: ifconfig

3 Puede pedir confirmacion, escribir yes, enter]

4 Acceder a alguna carpeta:

Tarjeta de memoria: >cd /media/card

Home: >cd

5 Crear carpeta donde alojar los archivos, y acceder a ella

6 Copiar codigo de uno de los archivos en la pc, con Ctrl+C

7 Para cada archivo del lado del servidor: Crear y acceder al archivo donde se va a pegar el codigo, con

>vi aws.c

8 Pegar codigo del programa, con Ctrl+Shift+V

9 Ir al principio del archivo, y agregar los primeros dos caracteres que el programa vi borrar al pegar.

10 Guardar los cambios y Salir con: ESC, Dos puntos, x, Enter

11 Compilar el programa con el comando **make**.

12 Ejecutar: **./s 6020**

Para compilar y ejecutar el servidor

En el lado del cliente se debe compilar el cliente por consola a través del comando **make** previamente ingresando a la carpeta.

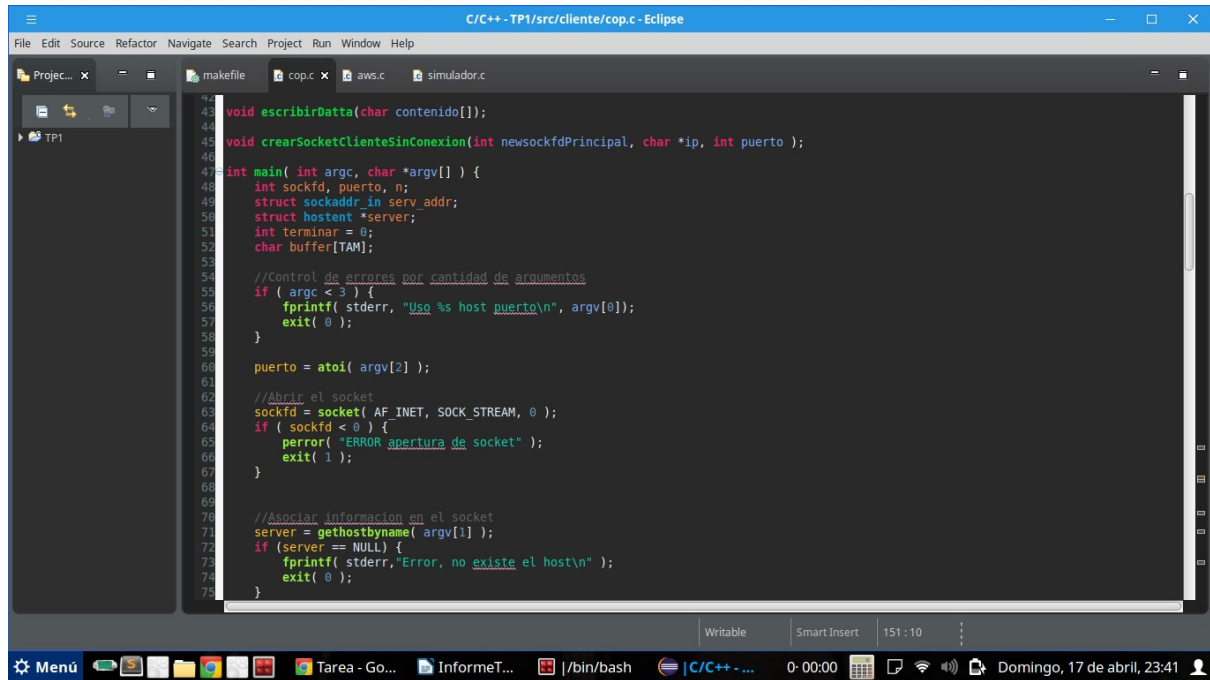
Ejecutar el cliente con: **>./c 192.168.1.100 6020**

donde 192.168.1.100 es la ip de la placa Galileo y se la puede averiguar por ejemplo, mirando la lista DHCP del router.

Una vez conectado el cliente, aparecerá un prompt para ingresar los comandos explicados en el apartado anterior.

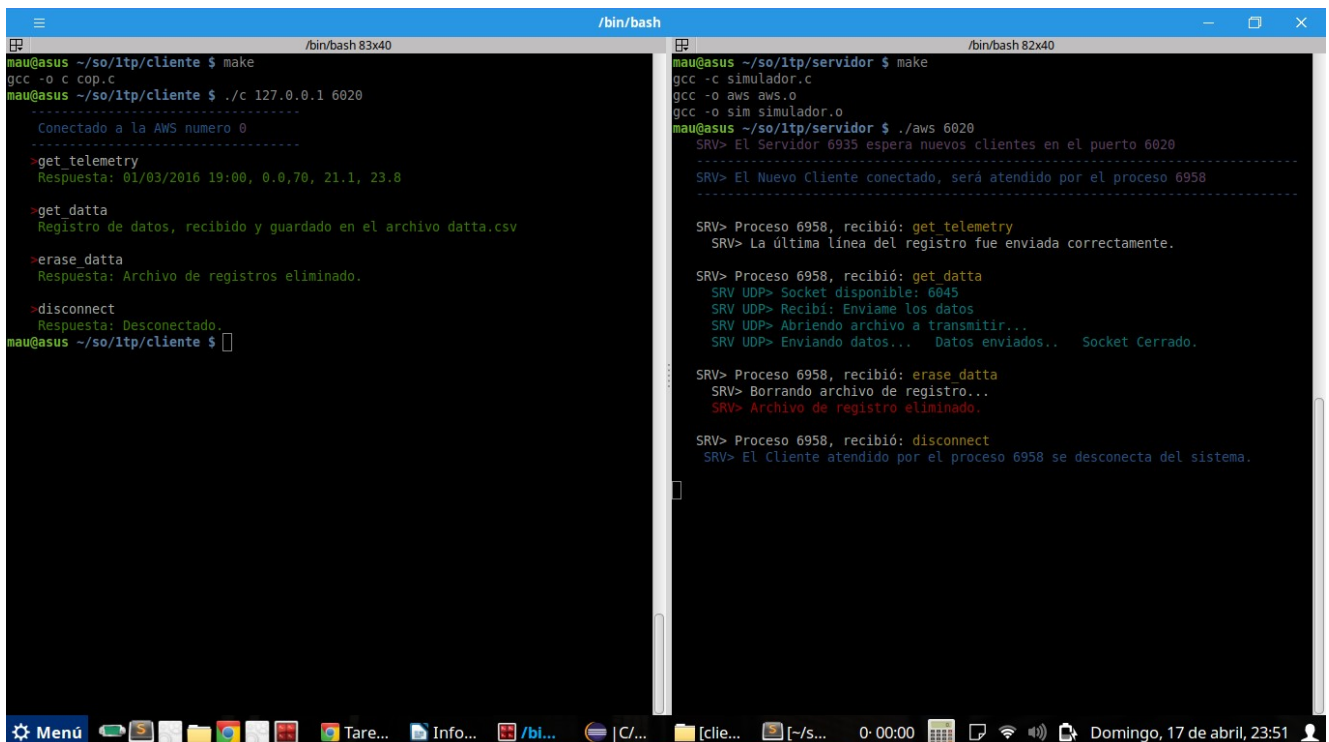
Implementación y Resultados

Para programar las diferentes aplicaciones se utilizó el programa Eclipse C/C++:



```
43 void escribirDatta(char contenido[]);
44
45 void crearSocketClientesSinConexion(int newsockfdPrincipal, char *ip, int puerto );
46
47 int main( int argc, char *argv[] ) {
48     int sockfd, puerto, n;
49     struct sockaddr_in serv_addr;
50     struct hostent *server;
51     int terminar = 0;
52     char buffer[TAM];
53
54     //Control de errores por cantidad de argumentos
55     if ( argc < 3 ) {
56         fprintf( stderr, "Uso %s host puerto\n", argv[0] );
57         exit( 0 );
58     }
59
60     puerto = atoi( argv[2] );
61
62     //Abrir el socket
63     sockfd = socket( AF_INET, SOCK_STREAM, 0 );
64     if ( sockfd < 0 ) {
65         perror( "ERROR apertura de socket" );
66         exit( 1 );
67     }
68
69     //Asociar informacion en el socket
70     server = gethostbyname( argv[1] );
71     if ( server == NULL ) {
72         fprintf( stderr, "Error, no existe el host\n" );
73         exit( 0 );
74     }
75 }
```

Para compilar se utilizó el programa **gcc** y se automatizó el proceso con **make**.



```
mau@asus ~/so/ltp/cliente $ make
gcc -o c cop.c
mau@asus ~/so/ltp/cliente $ ./c 127.0.0.1 6020
Conectado a la AWS numero 0
-----
>get telemetry
Respuesta: 01/03/2016 19:00, 0.0,70, 21.1, 23.8

>get datta
Registro de datos, recibido y guardado en el archivo datta.csv

>erase datta
Respuesta: Archivo de registros eliminado.

>disconnect
Respuesta: Desconectado.
mau@asus ~/so/ltp/cliente $

mau@asus ~/so/ltp/servidor $ make
gcc -c simulador.c
gcc -o aws aws.o
gcc -o sim simulador.o
mau@asus ~/so/ltp/servidor $ ./aws 6020
SRV> El Servidor 6935 espera nuevos clientes en el puerto 6020
-----
SRV> El Nuevo Cliente conectado, será atendido por el proceso 6958
-----
SRV> Proceso 6958, recibió: get telemetry
SRV> La última línea del registro fue enviada correctamente.
-----
SRV> Proceso 6958, recibió: get datta
SRV UDP> Socket disponible: 6045
SRV UDP> Recibi: Envíame los datos
SRV UDP> Abriendo archivo a transmitir...
SRV UDP> Enviando datos... Datos enviados... Socket Cerrado.
-----
SRV> Proceso 6958, recibió: erase datta
SRV> Borrando archivo de registro...
SRV> Archivo de registro eliminado.
-----
SRV> Proceso 6958, recibió: disconnect
SRV> El Cliente atendido por el proceso 6958 se desconecta del sistema.
```

Conclusiones

Se aprendió a utilizar la herramienta de Sockets para implementar una comunicación entre un cliente y servidor.