



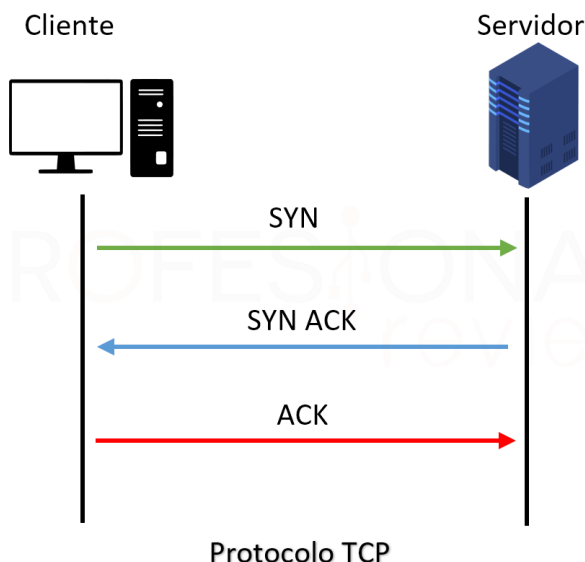
Servicio TCP de la capa de transporte

TCP (Transmission Control Protocol) es un protocolo de comunicación de datos utilizado en la capa de transporte del modelo OSI (Open Systems Interconnection) y en el modelo TCP/IP. Es un protocolo orientado a la conexión que proporciona una transmisión de datos fiable y en orden, asegurando que los datos enviados desde una máquina lleguen correctamente a su destino.

UDP (User Datagram Protocol) y TCP son los protocolos básicos de nivel de transporte para realizar conexiones entre sistemas principales de Internet. TCP y UDP permiten que los programas envíen mensajes a las aplicaciones de otros sistemas principales y reciban mensajes de dichas aplicaciones. Cuando una aplicación envía a la capa de transporte una petición de envío de un mensaje, UDP y TCP dividen la información en paquetes, añaden una cabecera de paquete incluida la dirección de destino y envían la información a la capa de red para su proceso adicional. TCP y UDP utilizan puertos de protocolo en el sistema principal para identificar el destino específico del mensaje.

Las aplicaciones y los protocolos de nivel superior utilizan UDP para realizar conexiones de datagrama y TCP para realizar conexiones de corriente. La interfaz de sockets de sistema operativo implementa estos protocolos.

- TCP proporciona entrega continua fiable de datos entre sistemas principales de Internet.
- Definiciones de campo de cabecera de TCP.



Obviamente interviene en la capa de transporte tanto de OSI como TCP/IP y es un protocolo orientado a la conexión a diferencia de IP. Esto significa que los dos equipos que intervienen deben aceptar la conexión antes de efectuar el intercambio de datos. Esto es muy importante para asegurar que los datos lleguen y toda la comunicación de estas características:

- La comunicación empieza y finaliza de forma cordial y sin roturas.
- Asegura el transporte fiable de datos para que todos lleguen al destino.
- Ordena los segmentos de datos cuando llevan a través del protocolo IP.
- Los datos pueden formar segmentos de longitud variable.



- Puede monitorizar el flujo de datos para no saturar la red, lo que vendría siendo un QoS (Quality of Service).
- Permite circular datos simultáneamente, aunque vengan de distintas funciones, lo que se llama multiplexación.

Desarrollo:

Inicialmente reamos un proyecto en Visual Studio Code, basta con crear una carpeta y dos scripts con extensión .js para esta práctica se nombraron de la siguiente manera:

- P_tcp_cliente.js
- P_tcp_servidor.js

Comenzaremos por comprender que conforma el servidor:

El módulo net en Node.js proporciona una API para crear y manejar tanto servidores como clientes de red. Permite trabajar con conexiones TCP (Transmission Control Protocol) y algunas funcionalidades básicas de conexiones en red, permite crear un clientes y servidores TCP.

El módulo net en Node.js ofrece soporte para IPC (Inter-Process Communication) y conexiones de red TCP. Proporciona clases y métodos para crear y manejar servidores (net.Server), clientes/sockets (net.Socket), y listas de bloqueo (net.BlockList). También incluye utilidades para gestionar conexiones, direcciones IP, y eventos relacionados con la red.

Puedes conocer las funcionalidades que ofrece el módulo en la documentación de node: <https://nodejs.org/api/net.html#new-netserveroptions-connectionlistener>

Ahora seguimos definiendo la lógica que implica la práctica comenzando con el servidor:

Primero, se importa el módulo net para utilizar las funcionalidades de red TCP. Luego, se crea un servidor TCP utilizando net.createServer(). Se define un manejador de eventos para la conexión ('connection'), donde se configuran manejadores para cuando se recibe un mensaje de un cliente ('data'), cuando la conexión se cierra ('close') y cuando ocurre un error ('error'). En el evento 'data', el servidor imprime el mensaje del cliente y responde con 'Mensaje recibido'. Finalmente, el servidor comienza a escuchar en el puerto 8080 y se imprime un mensaje confirmando que el servidor está escuchando.

Teniendo el siguiente fragmento de código en P_tcp_servidor.js:

```
const net = require('net');

const server = net.createServer();

server.on('connection', (socket)=>{
  socket.on('data', (data)=>{
    console.log('\nEl cliente ' + socket.remoteAddress + ":" +
    socket.remotePort + "dice: " + data);
    socket.write('Mensaje recibido');
  });

  socket.on('close', ()=>{
    console.log('Comunicación finalizada');
  });
});
```



```
});  
  
socket.on('error', (err)=>{  
    console.log(err.message);  
});  
});  
  
server.listen(8080, ()=>{  
    console.log('servidor esta escuchando al puerto', server.address().port);  
});
```

Ahora continuamos con el cliente primero conociendo lo que realiza nuestro código:

Primero, se importa el módulo net para utilizar las funcionalidades de red TCP y readline-sync para permitir la entrada del usuario desde la línea de comandos. Luego, se define la configuración del cliente en el objeto configuracion, especificando el puerto y la dirección del servidor. A continuación, se crea una conexión TCP utilizando net.createConnection(configuracion).

Antes de proceder al código primero debemos de instalar la biblioteca readline-sync para permitirnos la entrada en línea de comandos de forma síncrona con el siguiente comando en la terminal:

```
npm install readline-sync
```

Posterior a ello agregamos la lógica a nuestro archivo P_tcp_cliente.js:

```
const net = require('net');  
const readline = require('readline-sync');  
  
const configuracion = {  
    port: 8080,  
    host: '127.0.0.1'  
};  
  
const cliente = net.createConnection(configuracion);  
  
cliente.on('connect', ()=>{  
    console.log('Conexión exitosa');  
    sendLine();  
});  
  
cliente.on('data', (data)=>{  
    console.log('El servidor dice:' + data);  
    sendLine();  
});  
  
cliente.on('error', (err)=>{  
    console.log(err.message);  
});  
  
function sendLine() {  
    var line = readline.question('\ndigita tu mensaje: \t');  
    if (line == "0") {  
        cliente.end();  
    }  
}
```



```
}else{
    cliente.write(line);
}
}
```

Una vez establecida la lógica se deben de abrir dos terminales, una para la ejecución del servidor TCP y otra para la ejecución del cliente TCP como se muestra a continuación:

```
PROBLEMS 20 OUTPUT DEBUG CONSOLE TERMINAL PORTS
P [redacted] caciones en red\TCP> node .\p_tcp_servidor.js
servidor esta escuchando al puerto 8080
[redacted]
[redacted] caciones en red\TCP> node .\p_tcp_cliente.js
Conexión exitosa
digita tu mensaje: [redacted]
```

Se debe tener una salida como la que se presenta a continuación:

```
servidor esta escuchando al puerto 8080
El cliente ::ffff:127.0.0.1:61653dice: Hola esta es la practica de TCP
El cliente ::ffff:127.0.0.1:61653dice: Aqui termina la practica TCP
El cliente ::ffff:127.0.0.1:61653dice: Adios
read ECONNRESET
Comunicación finalizada
[redacted]
Conexión exitosa
digita tu mensaje: Hola esta es la practica de TCP
El servidor dice:Mensaje recibido
digita tu mensaje: Aqui termina la practica TCP
El servidor dice:Mensaje recibido
digita tu mensaje: Adios
El servidor dice:Mensaje recibido
```

También se puede hacer el análisis de los datos recibidos a través de wireshark como se muestra a continuación:

No.	Time	Source	Destination	Protocol	Length	Info
3	2.281239	127.0.0.1	127.0.0.1	TCP	75	61653 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=10233 Len=31 [TCP segment of a reassembled PDU]
4	2.281289	127.0.0.1	127.0.0.1	TCP	44	8080 → 61653 [ACK] Seq=1 Ack=32 Win=10233 Len=0
5	2.281925	127.0.0.1	127.0.0.1	TCP	60	8080 → 61653 [PSH, ACK] Seq=1 Ack=32 Win=10233 Len=16 [TCP segment of a reassembled PDU]
6	2.281939	127.0.0.1	127.0.0.1	TCP	44	61653 → 8080 [ACK] Seq=32 Ack=17 Win=10233 Len=0
9	12.953298	127.0.0.1	127.0.0.1	TCP	72	61653 → 8080 [PSH, ACK] Seq=32 Ack=17 Win=10233 Len=28 [TCP segment of a reassembled PDU]
10	12.953334	127.0.0.1	127.0.0.1	TCP	44	8080 → 61653 [ACK] Seq=17 Ack=60 Win=10233 Len=0
11	12.953733	127.0.0.1	127.0.0.1	TCP	60	8080 → 61653 [PSH, ACK] Seq=17 Ack=60 Win=10233 Len=16 [TCP segment of a reassembled PDU]
12	12.953748	127.0.0.1	127.0.0.1	TCP	44	61653 → 8080 [ACK] Seq=60 Ack=33 Win=10233 Len=0
15	17.017211	127.0.0.1	127.0.0.1	TCP	49	61653 → 8080 [PSH, ACK] Seq=60 Ack=33 Win=10233 Len=5 [TCP segment of a reassembled PDU]
16	17.017239	127.0.0.1	127.0.0.1	TCP	44	8080 → 61653 [ACK] Seq=33 Ack=65 Win=10233 Len=0
17	17.017529	127.0.0.1	127.0.0.1	TCP	60	8080 → 61653 [PSH, ACK] Seq=33 Ack=65 Win=10233 Len=16 [TCP segment of a reassembled PDU]
18	17.017544	127.0.0.1	127.0.0.1	TCP	44	61653 → 8080 [ACK] Seq=65 Ack=49 Win=10233 Len=0
19	18.378454	127.0.0.1	127.0.0.1	TCP	44	61653 → 8080 [RST, ACK] Seq=65 Ack=49 Win=0 Len=0

Teniendo una vista detallada de los mensajes enviados como se muestra a continuación:

Frame 3: 75 bytes on wire (600 bits), 75 bytes captured (600 bits) on interface \Device\NPF_{...} Loopback, 1 id

Null/loopback

Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

Transmission Control Protocol, Src Port: 61653, Dst Port: 8080, Seq: 1, Ack: 1, Len: 31

Source Port: 61653

Destination Port: 8080

[Stream index: 1]

[Conversation completeness: Incomplete (44)]

[TCP Segment Len: 31]

Sequence Number: 1 (relative sequence number)

Sequence Number (raw): 1278991574

[Next Sequence Number: 32 (relative sequence number)]

Acknowledgment Number: 1 (relative ack number)

Acknowledgment number (raw): 1188796392

0101 ... = Header Length: 20 bytes (5)

Flags: 0x018 (PSH, ACK)

Window: 10233

[Calculated window size: 10233]

[Window size scaling factor: -1 (unknown)]

Checksum: 0x1fbd [unverified]

[Checksum Status: Unverified]

Urgent Pointer: 0

[Timestamps]

[SEQ/ACK analysis]

TCP payload (31 bytes)

TCP segment data (31 bytes)

0000 02 00 00 00 45 00 00 47 75 c2 40 00 00 00 00 ...E:G u@...

0010 7f 00 00 01 7f 00 00 01 f0 d5 1f 90 4b c1 ca d6 ...K...

0020 46 db 97 e8 50 18 27 f9 1f bd 00 00 48 6f 6c 61 ...P.'...Hla

0030 20 65 73 74 61 20 65 73 20 6c 61 20 70 72 61 63 ...esta es la prac

0040 74 69 63 61 20 64 65 20 54 43 50 ...tica de TCP