



Instituto Politécnico Nacional

IPN

Escuela Superior de Cómputo

ESCOM

Academia de Redes Aplicaciones para comunicaciones
en red

6CV2

Práctica 5

“Servicio TCP de la capa de transporte en Java”

Alumna:

Navarrete Becerril Sharon Anette

Fecha de entrega: “Lunes, 21-Octubre-2024”

Profesor: Ojeda Santillan Rodrigo

OBJETIVO

Desarrollar una comprensión técnica y práctica del Protocolo de Control de Transmisión (TCP) a través de la implementación de un servidor y un cliente utilizando Node.js, con el fin de observar el comportamiento y flujo de datos en una red TCP, incluyendo el análisis de tráfico mediante Wireshark.

INTRODUCCIÓN

El Protocolo de Control de Transmisión (TCP) es un protocolo fundamental en la capa de transporte de los modelos OSI y TCP/IP. A diferencia de otros protocolos como UDP, TCP está orientado a la conexión y garantiza una transmisión fiable y en orden de los datos. Su capacidad para gestionar conexiones, realizar seguimiento de la entrega de datos y controlar el flujo de información lo convierte en una opción preferida para aplicaciones que requieren integridad y consistencia en la transmisión de datos. En este proyecto, se implementará un servidor y un cliente TCP utilizando el módulo `net` de Node.js para observar cómo funcionan las conexiones TCP y analizar el tráfico de red generado utilizando la herramienta Wireshark.

DESARROLLO

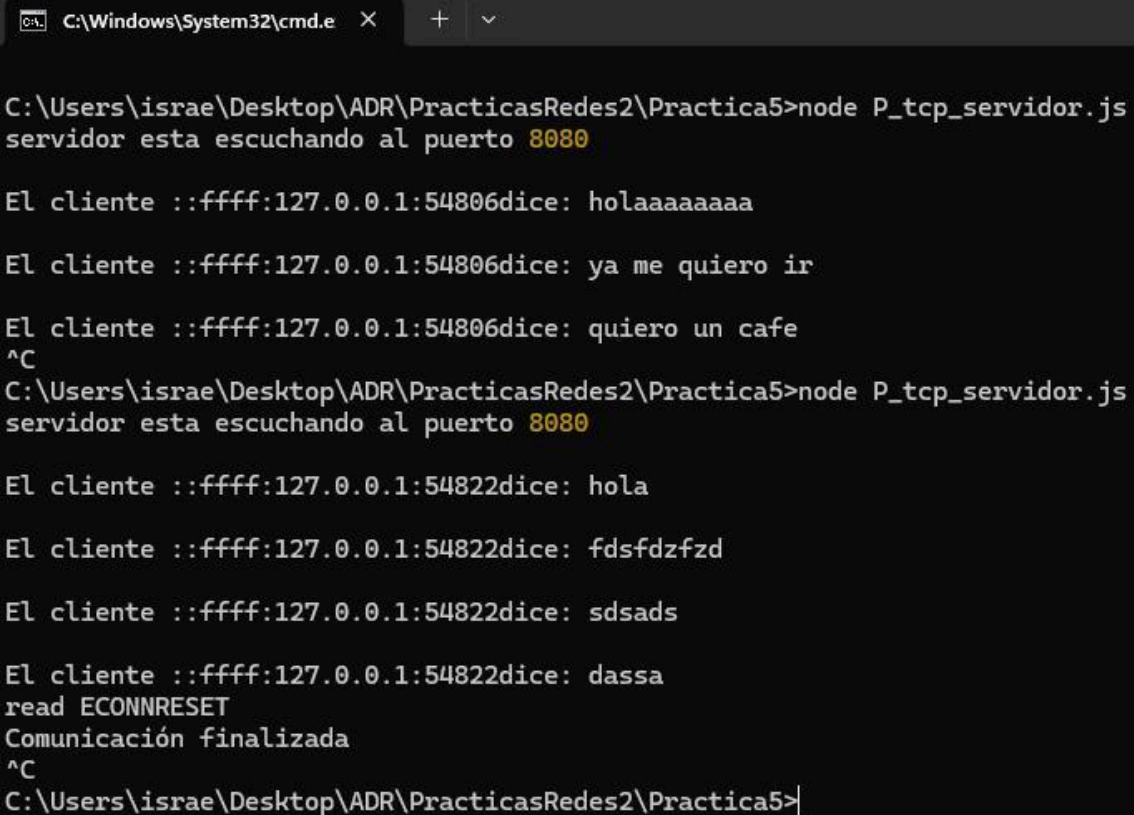
Para poner en práctica el concepto de TCP, se crearon dos scripts en Node.js: uno para el servidor (`P_tcp_servidor.js`) y otro para el cliente (`P_tcp_cliente.js`). El servidor utiliza el módulo `net` para crear un servicio TCP que escucha en el puerto 8080. Una vez que un cliente establece la conexión, el servidor gestiona los mensajes entrantes y envía respuestas confirmando la recepción de los datos.

Código del servidor:

```
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);
});
```



```
C:\Windows\System32\cmd.e  X  +  v

C:\Users\israe\Desktop\ADR\PracticasRedes2\Practica5>node P_tcp_servidor.js
servidor esta escuchando al puerto 8080

El cliente ::ffff:127.0.0.1:54806dice: holaaaaaaaa
El cliente ::ffff:127.0.0.1:54806dice: ya me quiero ir
El cliente ::ffff:127.0.0.1:54806dice: quiero un cafe
^C
C:\Users\israe\Desktop\ADR\PracticasRedes2\Practica5>node P_tcp_servidor.js
servidor esta escuchando al puerto 8080

El cliente ::ffff:127.0.0.1:54822dice: hola
El cliente ::ffff:127.0.0.1:54822dice: fdsfdzfdz
El cliente ::ffff:127.0.0.1:54822dice: sdsads
El cliente ::ffff:127.0.0.1:54822dice: dassa
read ECONNRESET
Comunicación finalizada
^C
C:\Users\israe\Desktop\ADR\PracticasRedes2\Practica5>
```

Cliente

Por otro lado, el cliente también utiliza el módulo **net** junto con la biblioteca **readline-sync** para permitir la entrada de datos desde la línea de comandos, y posteriormente envía estos datos al servidor. Ambos scripts permiten observar en tiempo real el intercambio de datos entre el cliente y el servidor. Para obtener una visión más detallada de los paquetes de datos transmitidos

Código del cliente:

```
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);
    }
}

```

```

C:\Windows\System32\cmd.e X + v
C:\Users\israe\Desktop\ADR\PracticasRedes2\Practica5>node P_tcp_cliente.js
Conexión exitosa

digita tu mensaje:      holaaaaaaaa
El servidor dice:Mensaje recibido

digita tu mensaje:      ya me quiero ir
El servidor dice:Mensaje recibido

digita tu mensaje:      quiero un cafe
El servidor dice:Mensaje recibido

digita tu mensaje:      ^C
C:\Users\israe\Desktop\ADR\PracticasRedes2\Practica5>node P_tcp_cliente.js
Conexión exitosa

digita tu mensaje:      hola
El servidor dice:Mensaje recibido

digita tu mensaje:      fdsfdzfzd
El servidor dice:Mensaje recibido

digita tu mensaje:      sdsads
El servidor dice:Mensaje recibido

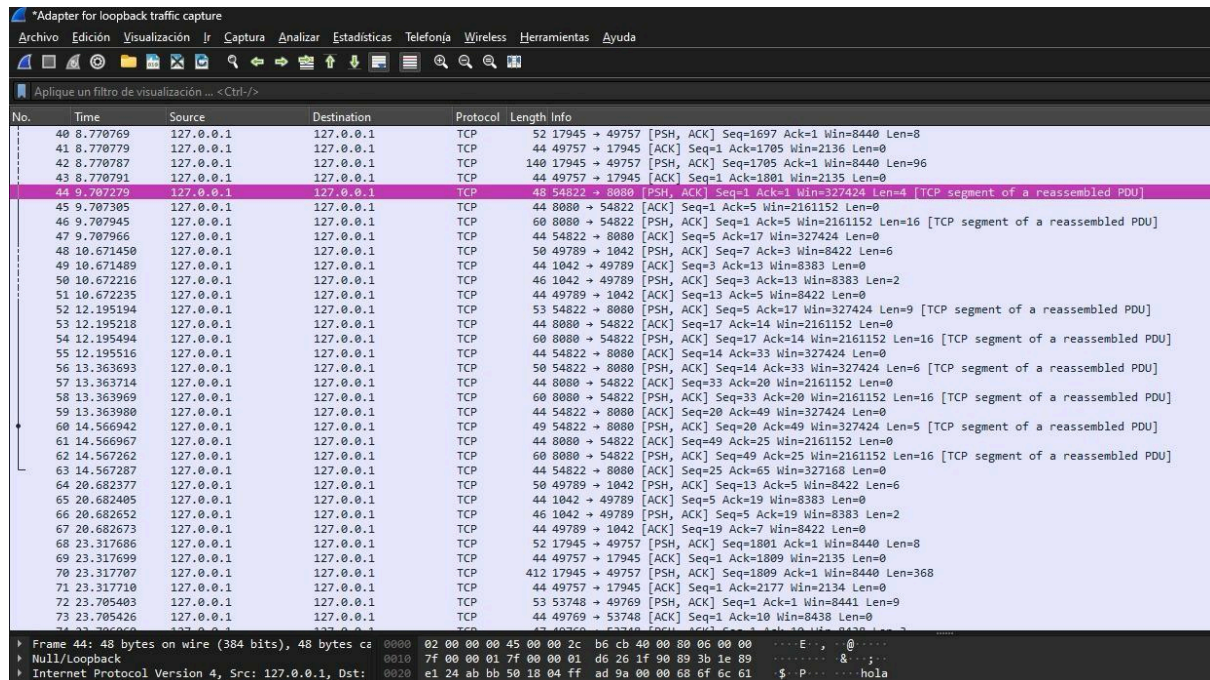
digita tu mensaje:      dassa
El servidor dice:Mensaje recibido

digita tu mensaje:      ^C
C:\Users\israe\Desktop\ADR\PracticasRedes2\Practica5>

```

Análisis de tráfico con Wireshark

Se utilizó la herramienta Wireshark, que permite analizar los paquetes TCP y evaluar la calidad de la transmisión.



No.	Time	Source	Destination	Protocol	Length	Info
40	8.770769	127.0.0.1	127.0.0.1	TCP	52	17945 → 49757 [PSH, ACK] Seq=1697 Ack=1 Win=8440 Len=8
41	8.770779	127.0.0.1	127.0.0.1	TCP	44	49757 → 17945 [ACK] Seq=1 Ack=1705 Win=2136 Len=0
42	8.770787	127.0.0.1	127.0.0.1	TCP	140	17945 → 49757 [PSH, ACK] Seq=1705 Ack=1 Win=8440 Len=96
43	8.770791	127.0.0.1	127.0.0.1	TCP	44	49757 → 17945 [ACK] Seq=1 Ack=1801 Win=2135 Len=0
44	9.702279	127.0.0.1	127.0.0.1	TCP	48	54822 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=327424 Len=4 [TCP segment of a reassembled PDU]
45	9.707395	127.0.0.1	127.0.0.1	TCP	44	8080 → 54822 [ACK] Seq=1 Ack=5 Win=2161152 Len=0
46	9.707945	127.0.0.1	127.0.0.1	TCP	60	8080 → 54822 [PSH, ACK] Seq=1 Ack=5 Win=2161152 Len=16 [TCP segment of a reassembled PDU]
47	9.707966	127.0.0.1	127.0.0.1	TCP	44	54822 → 8080 [ACK] Seq=5 Ack=17 Win=327424 Len=0
48	10.671450	127.0.0.1	127.0.0.1	TCP	50	49789 → 1042 [PSH, ACK] Seq=7 Ack=3 Win=8422 Len=6
49	10.671489	127.0.0.1	127.0.0.1	TCP	44	1042 → 49789 [ACK] Seq=3 Ack=13 Win=8383 Len=0
50	10.672216	127.0.0.1	127.0.0.1	TCP	46	1042 → 49789 [PSH, ACK] Seq=3 Ack=13 Win=8383 Len=2
51	10.672235	127.0.0.1	127.0.0.1	TCP	44	49789 → 1042 [ACK] Seq=13 Ack=5 Win=8422 Len=0
52	12.195194	127.0.0.1	127.0.0.1	TCP	53	54822 → 8080 [PSH, ACK] Seq=5 Ack=17 Win=327424 Len=9 [TCP segment of a reassembled PDU]
53	12.195218	127.0.0.1	127.0.0.1	TCP	44	8080 → 54822 [ACK] Seq=17 Ack=14 Win=2161152 Len=0
54	12.195494	127.0.0.1	127.0.0.1	TCP	60	8080 → 54822 [PSH, ACK] Seq=17 Ack=14 Win=2161152 Len=16 [TCP segment of a reassembled PDU]
55	12.195516	127.0.0.1	127.0.0.1	TCP	44	54822 → 8080 [ACK] Seq=14 Ack=33 Win=327424 Len=0
56	13.363693	127.0.0.1	127.0.0.1	TCP	50	54822 → 8080 [PSH, ACK] Seq=14 Ack=33 Win=327424 Len=6 [TCP segment of a reassembled PDU]
57	13.363714	127.0.0.1	127.0.0.1	TCP	44	8080 → 54822 [ACK] Seq=33 Ack=20 Win=2161152 Len=0
58	13.363969	127.0.0.1	127.0.0.1	TCP	60	8080 → 54822 [PSH, ACK] Seq=33 Ack=20 Win=2161152 Len=16 [TCP segment of a reassembled PDU]
59	13.363980	127.0.0.1	127.0.0.1	TCP	44	54822 → 8080 [ACK] Seq=20 Ack=49 Win=327424 Len=0
60	14.566942	127.0.0.1	127.0.0.1	TCP	49	54822 → 8080 [PSH, ACK] Seq=20 Ack=49 Win=327424 Len=5 [TCP segment of a reassembled PDU]
61	14.566967	127.0.0.1	127.0.0.1	TCP	44	8080 → 54822 [ACK] Seq=49 Ack=25 Win=2161152 Len=0
62	14.567262	127.0.0.1	127.0.0.1	TCP	60	8080 → 54822 [PSH, ACK] Seq=49 Ack=25 Win=2161152 Len=16 [TCP segment of a reassembled PDU]
63	14.567287	127.0.0.1	127.0.0.1	TCP	44	54822 → 8080 [ACK] Seq=25 Ack=65 Win=327168 Len=0
64	20.682377	127.0.0.1	127.0.0.1	TCP	50	49789 → 1042 [PSH, ACK] Seq=13 Ack=5 Win=8422 Len=6
65	20.682405	127.0.0.1	127.0.0.1	TCP	44	1042 → 49789 [ACK] Seq=5 Ack=19 Win=8383 Len=0
66	20.682652	127.0.0.1	127.0.0.1	TCP	46	1042 → 49789 [PSH, ACK] Seq=5 Ack=19 Win=8383 Len=2
67	20.682673	127.0.0.1	127.0.0.1	TCP	44	49789 → 1042 [ACK] Seq=19 Ack=7 Win=8422 Len=0
68	23.317686	127.0.0.1	127.0.0.1	TCP	52	17945 → 49757 [PSH, ACK] Seq=1801 Ack=1 Win=8440 Len=8
69	23.317699	127.0.0.1	127.0.0.1	TCP	44	49757 → 17945 [ACK] Seq=1 Ack=1809 Win=2135 Len=0
70	23.317707	127.0.0.1	127.0.0.1	TCP	412	17945 → 49757 [PSH, ACK] Seq=1809 Ack=1 Win=8440 Len=368
71	23.317710	127.0.0.1	127.0.0.1	TCP	44	49757 → 17945 [ACK] Seq=1 Ack=2177 Win=2134 Len=0
72	23.705403	127.0.0.1	127.0.0.1	TCP	53	53748 → 49769 [PSH, ACK] Seq=1 Ack=1 Win=8441 Len=9
73	23.705426	127.0.0.1	127.0.0.1	TCP	44	49769 → 53748 [ACK] Seq=1 Ack=10 Win=8438 Len=0

Frame 44: 48 bytes on wire (384 bits), 48 bytes captured on interface 0, 48 bytes from 127.0.0.1 to 127.0.0.1 on interface 0

Ethernet II, Src: Adapter for loopback traffic capture (000000000000), Dst: 127.0.0.1 (010000000000)

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

TCP, Seq=54822, Ack=8080, Win=327424, Len=4

PSH, ACK Seq=1 Ack=1 Win=327424 Len=4

Application/javascript

E...@...&...\$-P...hola

CONCLUSIONES

Navarrete Becerril Sharon Anette:

Implementar TCP en Node.js es eficiente y flexible: Node.js, con su módulo **net**, facilita la creación de aplicaciones de red TCP, permitiendo a los desarrolladores crear servidores y clientes TCP de manera sencilla. Además, proporciona flexibilidad para manejar errores y gestionar múltiples conexiones de manera eficiente.

Bibliografía

Stevens, W. Richard. Programación en red con Unix: La API de Sockets. Addison-Wesley Professional, 2003.

Forouzan, Behrouz A. Comunicaciones de Datos y Redes. McGraw-Hill, 2012. Tanenbaum, Andrew S., y David J. Wetherall. Redes de Computadoras. Pearson, 2010.

Comer, Douglas E. Internetworking con TCP/IP Volumen Uno. Prentice Hall, 2006. Kurose, James F., y Keith W. Ross. Redes de Computadoras: Un Enfoque Descendente. Pearson, 2017.

Beazley, David, y Brian K. Jones. Python Cookbook: Recetas para Dominar Python 3. O'Reilly Media, 2013. Begg, A. Sockets TCP/IP en C: Guía Práctica para Programadores. Morgan Kaufmann, 2000