UNIVERSIDAD DE LOS ANDES

DEPARTAMENTO DE INGENIERIA DE SISTEMAS Y COMPUTACIÓN

LABORATORIO #3: ANALISIS CAPA DE TRANSPORTE Y SOCKETS

ISIS3204 – INFRAESTRUCTURA DE COMUNICACIONES

Yuri Pinto

Natalia Quiroga

Grupo 2

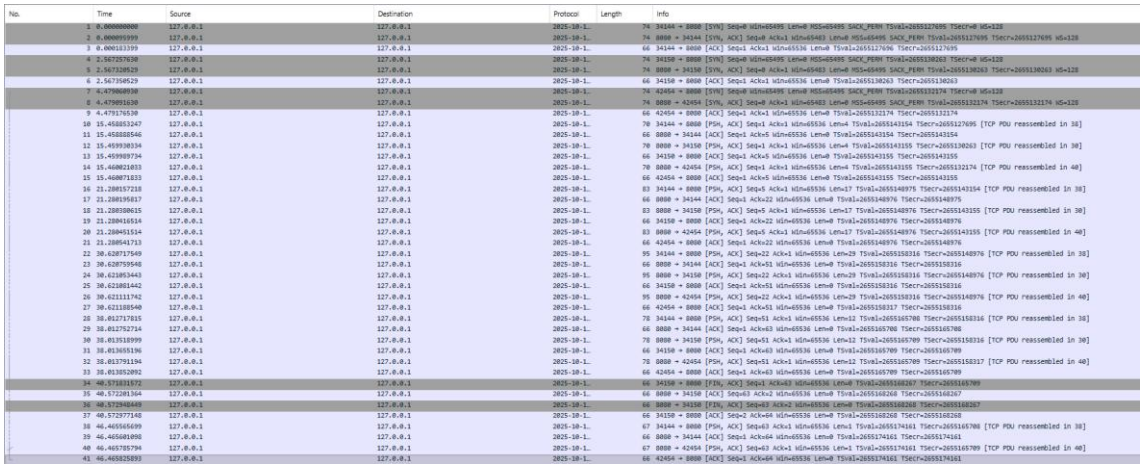
Sebastián Martínez Arias - 202312210

Esteban Alejandro Hernández Sulvara – 202316637

Santiago Gómez Ordoñez - 202315097

Contenido

ANÁLISIS DETALLADO POR ETAPAS DE COMUNICACIÓN TCP PRE-INFORME



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	2025-10-1…		74 34144 → 8080 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2655127695 TSecr=0 WS=128
2	0.000095999	127.0.0.1	127.0.0.1	2025-10-1…		74 8080 → 34144 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=2655127695 TSecr=2655127695 WS=128
3	0.000163399	127.0.0.1	127.0.0.1	2025-10-1…		66 34144 → 8080 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2655127696 TSecr=2655127695
4	2.567257520	127.0.0.1	127.0.0.1	2025-10-1…		74 34150 → 8080 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2655130263 TSecr=0 WS=128
5	2.567328529	127.0.0.1	127.0.0.1	2025-10-1…		74 8080 → 34150 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=2655130263 TSecr=2655130263 WS=128
6	2.567350529	127.0.0.1	127.0.0.1	2025-10-1…		66 34150 → 8080 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2655130263 TSecr=2655130263
7	4.479060930	127.0.0.1	127.0.0.1	2025-10-1…		74 42454 → 8080 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2655132174 TSecr=0 WS=128
8	4.479091630	127.0.0.1	127.0.0.1	2025-10-1…		74 8080 → 42454 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=2655132174 TSecr=2655132174 WS=128
9	4.479176530	127.0.0.1	127.0.0.1	2025-10-1…		66 42454 → 8080 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2655132174 TSecr=2655132174
10	15.458053247	127.0.0.1	127.0.0.1	2025-10-1…		70 34144 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=4 TSval=2655143154 TSecr=2655127695 [TCP PDU reassembled in 38]
11	15.458088546	127.0.0.1	127.0.0.1	2025-10-1…		66 8080 → 34144 [ACK] Seq=1 Ack=5 Win=65536 Len=0 TSval=2655143154 TSecr=2655143154
12	15.459930934	127.0.0.1	127.0.0.1	2025-10-1…		70 8080 → 34150 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=4 TSval=2655143155 TSecr=2655130263 [TCP PDU reassembled in 38]
13	15.459990734	127.0.0.1	127.0.0.1	2025-10-1…		66 34150 → 8080 [ACK] Seq=1 Ack=5 Win=65536 Len=0 TSval=2655143155 TSecr=2655143155
14	15.460021033	127.0.0.1	127.0.0.1	2025-10-1…		70 8080 → 42454 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=4 TSval=2655143155 TSecr=2655132174 [TCP PDU reassembled in 40]
15	15.460071833	127.0.0.1	127.0.0.1	2025-10-1…		66 42454 → 8080 [ACK] Seq=1 Ack=5 Win=65536 Len=0 TSval=2655143155 TSecr=2655143155
16	21.288157218	127.0.0.1	127.0.0.1	2025-10-1…		83 34144 → 8080 [PSH, ACK] Seq=5 Ack=1 Win=65536 Len=17 TSval=2655148976 TSecr=2655143154 [TCP PDU reassembled in 38]
17	21.288195817	127.0.0.1	127.0.0.1	2025-10-1…		66 8080 → 34144 [ACK] Seq=1 Ack=22 Win=65536 Len=0 TSval=2655148976 TSecr=2655148976
18	21.288500615	127.0.0.1	127.0.0.1	2025-10-1…		83 8080 → 34150 [PSH, ACK] Seq=5 Ack=1 Win=65536 Len=17 TSval=2655143155 TSecr=2655143155 [TCP PDU reassembled in 38]
19	21.288416514	127.0.0.1	127.0.0.1	2025-10-1…		66 34150 → 8080 [ACK] Seq=1 Ack=22 Win=65536 Len=0 TSval=2655148976 TSecr=2655148976
20	21.288451514	127.0.0.1	127.0.0.1	2025-10-1…		83 8080 → 42454 [PSH, ACK] Seq=5 Ack=1 Win=65536 Len=17 TSval=2655148976 TSecr=2655143155 [TCP PDU reassembled in 40]
21	21.288541713	127.0.0.1	127.0.0.1	2025-10-1…		66 42454 → 8080 [ACK] Seq=1 Ack=22 Win=65536 Len=0 TSval=2655148976 TSecr=2655148976
22	30.628717549	127.0.0.1	127.0.0.1	2025-10-1…		95 34144 → 8080 [PSH, ACK] Seq=22 Ack=1 Win=65536 Len=29 TSval=2655158316 TSecr=2655148976 [TCP PDU reassembled in 38]
23	30.628759548	127.0.0.1	127.0.0.1	2025-10-1…		66 8080 → 34144 [ACK] Seq=1 Ack=51 Win=65536 Len=0 TSval=2655158316 TSecr=2655158316
24	30.621053443	127.0.0.1	127.0.0.1	2025-10-1…		95 8080 → 34150 [PSH, ACK] Seq=22 Ack=1 Win=65536 Len=29 TSval=2655158316 TSecr=2655148976 [TCP PDU reassembled in 38]
25	30.621801442	127.0.0.1	127.0.0.1	2025-10-1…		66 34150 → 8080 [ACK] Seq=51 Win=65536 Len=0 TSval=2655158316 TSecr=2655158316
26	30.621111742	127.0.0.1	127.0.0.1	2025-10-1…		95 8080 → 42454 [PSH, ACK] Seq=22 Ack=1 Win=65536 Len=29 TSval=2655158316 TSecr=2655148976 [TCP PDU reassembled in 40]
27	30.621188540	127.0.0.1	127.0.0.1	2025-10-1…		66 42454 → 8080 [ACK] Seq=1 Ack=51 Win=65536 Len=0 TSval=2655158317 TSecr=2655158316
28	38.012717815	127.0.0.1	127.0.0.1	2025-10-1…		78 34144 → 8080 [PSH, ACK] Seq=51 Ack=1 Win=65536 Len=12 TSval=2655165708 TSecr=2655158316 [TCP PDU reassembled in 38]
29	38.012752714	127.0.0.1	127.0.0.1	2025-10-1…		66 8080 → 34144 [ACK] Seq=1 Ack=63 Win=65536 Len=0 TSval=2655165708 TSecr=2655165708
30	38.013518999	127.0.0.1	127.0.0.1	2025-10-1…		78 8080 → 34150 [PSH, ACK] Seq=51 Ack=1 Win=65536 Len=12 TSval=2655165709 TSecr=2655158316 [TCP PDU reassembled in 38]
31	38.013655196	127.0.0.1	127.0.0.1	2025-10-1…		66 34150 → 8080 [ACK] Seq=1 Ack=63 Win=65536 Len=0 TSval=2655165709 TSecr=2655165709
32	38.013791194	127.0.0.1	127.0.0.1	2025-10-1…		78 8080 → 42454 [PSH, ACK] Seq=51 Ack=1 Win=65536 Len=12 TSval=2655165709 TSecr=2655158317 [TCP PDU reassembled in 40]
33	38.013852092	127.0.0.1	127.0.0.1	2025-10-1…		66 42454 → 8080 [ACK] Seq=1 Ack=63 Win=65536 Len=0 TSval=2655165709 TSecr=2655165709
34	40.571811672	127.0.0.1	127.0.0.1	2025-10-1…		66 34150 → 8080 [FIN, ACK] Seq=1 Ack=63 Win=65536 Len=0 TSval=2655168267 TSecr=2655165709
35	40.572281364	127.0.0.1	127.0.0.1	2025-10-1…		66 8080 → 34150 [ACK] Seq=63 Ack=2 Win=65536 Len=0 TSval=2655168268 TSecr=2655168267
36	40.572940449	127.0.0.1	127.0.0.1	2025-10-1…		66 8080 → 34150 [FIN, ACK] Seq=63 Ack=2 Win=65536 Len=0 TSval=2655168268 TSecr=2655168267
37	40.572977148	127.0.0.1	127.0.0.1	2025-10-1…		66 34150 → 8080 [ACK] Seq=2 Ack=64 Win=65536 Len=0 TSval=2655168268 TSecr=2655168268
38	46.465565699	127.0.0.1	127.0.0.1	2025-10-1…		67 34144 → 8080 [PSH, ACK] Seq=63 Ack=1 Win=65536 Len=1 TSval=2655174161 TSecr=2655165708 [TCP PDU reassembled in 38]
39	46.465681098	127.0.0.1	127.0.0.1	2025-10-1…		66 8080 → 34144 [ACK] Seq=1 Ack=64 Win=65536 Len=0 TSval=2655174161 TSecr=2655174161
40	46.465785794	127.0.0.1	127.0.0.1	2025-10-1…		67 8080 → 42454 [PSH, ACK] Seq=63 Ack=1 Win=65536 Len=1 TSval=2655174161 TSecr=2655165709 [TCP PDU reassembled in 40]
41	46.465825893	127.0.0.1	127.0.0.1	2025-10-1…		66 42454 → 8080 [ACK] Seq=1 Ack=64 Win=65536 Len=0 TSval=2655174161 TSecr=2655174161

Ilustración 1: TCP Sin pérdida de Datos

1. ESTABLECIMIENTO DE CONEXIONES - THREE-WAY HANDSHAKE

Cliente 1 (Publicador/Suscriptor - Puerto 34144)

Dentro de las partes de secuencia 1 a 3, se demuestra de manera directa el funcionamiento del three-way handshake donde se hará la conexión entre los puertos 34144 y 8080. El primer paso demuestra la conexión inicial de SYN donde el número de secuencia está determinando la cantidad de bytes que se está transportando; una vez que realizamos esto, el siguiente paso es la conexión doble con el receptor donde este enviará un mensaje de recepción SYN y un acknowledge de que recibió el mensaje inicial de conexión. Finalmente, el último mensaje corresponde al acknowledge de conexión ACK para poder hacer una conexión directa.

Cliente 2 (Suscriptor - Puerto 34150)

En las secuencias 4 a 6, el Cliente 2 en el puerto 34150 inicia su three-way handshake con el broker (puerto 8080) enviando un segmento SYN. El broker responde con un SYN-ACK, confirmando la solicitud y estableciendo sus propios parámetros, a lo que el cliente responde con un ACK, completando así la conexión TCP de manera exitosa.

Cliente 3 (Suscriptor - Puerto 42454)

De manera similar, en las secuencias 7 a 9, el Cliente 3 en el puerto 42454 establece su conexión con el broker mediante el three-way handshake estándar, enviando un SYN, recibiendo un SYN-ACK del broker y finalizando con un ACK, lo que evidencia la confiabilidad del TCP, ya que las tres conexiones se establecieron exitosamente antes de cualquier transferencia de datos.

Primera Transferencia de Datos - Mensajes Cortos

El evento comienza con la secuencia 10, donde el Cliente 1 (34144) envía un mensaje inicial de 4 bytes al broker, probablemente de suscripción, utilizando las banderas PSH y ACK. Inmediatamente, el broker confirma la recepción con un ACK (secuencia 11) y luego redistribuye este mismo mensaje a ambos suscriptores (secuencias 12 y 14), quienes a su vez confirman la recepción con sus respectivos ACK (secuencias 13 y 15), demostrando el mecanismo de publicación/suscripción.

Segunda Transferencia - Mensajes de Eventos Deportivos

En la secuencia 16, el Cliente 1 envía un mensaje más largo de 17 bytes, que podría ser un evento deportivo como "Gol: EquipoA 1-0 32'". El broker confirma su recepción (secuencia 17) y procede a reenviar este mensaje a los suscriptores en los puertos 34150 (secuencia 18) y 42454 (secuencia 20), quienes nuevamente confirman con ACK (secuencias 19 y 21), mostrando la distribución eficiente de mensajes de mayor longitud.

Tercera Transferencia - Mensajes Más Extensos

Para la secuencia 22, el Cliente 1 envía un mensaje aún más detallado de 29 bytes. El broker confirma (secuencia 23) y realiza la redistribución a los suscriptores (secuencias 24 y 26), quienes envían sus acuses de recibo (secuencias 25 y 27), manejando sin problemas mensajes de tamaño creciente y manteniendo la confiabilidad de la entrega.

Cuarta Transferencia - Mensajes Cortos/Finales

En la secuencia 28, el Cliente 1 envía un mensaje de 12 bytes, posiblemente un comando de control como "DESCONECTAR". El broker lo reconoce (secuencia 29) y lo redistribuye a los suscriptores (secuencias 30 y 32), quienes confirman la recepción (secuencias 31 y 33), indicando una posible fase de finalización de la sesión.

Cierre de Conexión - Cliente 34150

El Cliente 2 (34150) inicia un cierre ordenado de su conexión enviando un FIN junto con un ACK (secuencia 34). El broker confirma la recepción de este FIN con un ACK (secuencia 35) y luego envía su propio FIN-ACK (secuencia 36), a lo que el cliente responde con un ACK final (secuencia 37), completando el cierre en ambas direcciones de manera correcta y ordenada.

Mensajes Finales - Byte Único

El broker lo reconoce (secuencia 39) y lo reenvía al suscriptor restante, el Cliente 3 (secuencia 40), quien confirma su recepción (secuencia 41), concluyendo la comunicación con un mensaje mínimo que asegura la entrega de todos los datos pendientes.

TABLA RESUMEN DE MENSAJES INFERIDOS

Paquete	Origen → Destino	Longitud	Propósito
10	34144 → 8080	4 bytes	Suscripción a partido
12,14	8080 → 34150,42454	4 bytes	Confirmación suscripción
16	34144 → 8080	17 bytes	Evento deportivo
18,20	Broker → Suscriptores	17 bytes	Redistribución
22	34144 → 8080	29 bytes	Evento detallado
24,26	Broker → Suscriptores	29 bytes	Redistribución

Paquete	Origen → Destino	Longitud	Propósito
28	34144 → 8080	12 bytes	Fin del partido
30,32	Broker → Suscriptores	12 bytes	Redistribución
38	34144 → 8080	1 byte	Desconexión
40	Broker → 42454	1 byte	Confirmación

CONCLUSIONES TÉCNICAS PARA EL INFORME

El análisis de la captura demuestra las ventajas de TCP mediante una confiabilidad absoluta, donde cada mensaje es confirmado con un ACK, garantizando la entrega. El orden de los datos está asegurado mediante secuencias incrementales (Seq 1→5→22→51→63), mientras que el control de flujo se maneja eficientemente con ventanas de 65536 bytes que permiten una transmisión óptima. Además, se implementa un cierre ordenado mediante el intercambio de FIN-ACK en ambas direcciones, y el broker demuestra capacidad para gestionar múltiples clientes simultáneamente, manejando tres conexiones de forma estable.

El patrón Publicador-Suscriptor se implementa correctamente con un publicador (34144) que envía mensajes a un broker (8080), quien los redistribuye a dos suscriptores (34150, 42454). Este esquema opera con una latencia mínima, donde los ACK se reciben en aproximadamente 0.0001 segundos, y mantiene un overhead controlado mediante el uso de cabeceras TCP junto con timestamps opcionales.

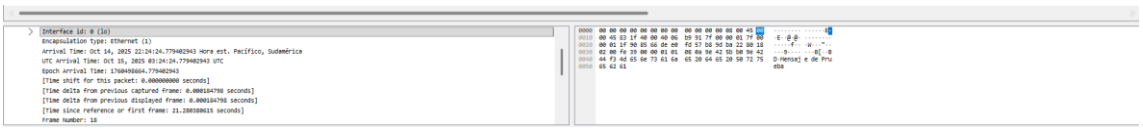


Ilustración 2: Banderas y Three-way handshake

ANÁLISIS DE BANDERAS TCP

La bandera SYN se utiliza exclusivamente para el establecimiento de conexión, como se observa en los paquetes 1, 4 y 7, donde los clientes solicitan iniciar el Three-Way Handshake sincronizando números de secuencia. Estas solicitudes se caracterizan por comenzar con Seq=0, anunciar un tamaño de ventana, negociar el Maximum Segment Size y no contener datos de aplicación.

Posteriormente, la bandera SYN + ACK es emitida por el servidor en respuesta, donde el broker acepta la conexión y confirma la recepción del SYN del cliente. Esta combinación incluye tanto el número de secuencia inicial del servidor como un campo Ack que confirma específicamente el SYN recibido del cliente.

La bandera ACK, presente en múltiples ejemplos de la captura, tiene el propósito de confirmar la recepción de datos, indicando el próximo byte esperado mediante un patrón

consistente donde el valor Ack se calcula como el Seq anterior más la Longitud del dato recibido, asegurando así la integridad de la secuencia.

Para la entrega inmediata de datos de aplicación, se utiliza la bandera PSH + ACK, que fuerza el envío inmediato de la información a la aplicación receptora. Esta bandera siempre se combina con ACK, contiene datos de aplicación (Len > 0) y se emplea en todos los mensajes con contenido real dentro de la comunicación.

Finalmente, la bandera FIN + ACK se utiliza para el cierre ordenado de la conexión, donde un extremo indica que no tiene más datos por enviar, mientras que el ACK confirma la recepción de los datos hasta ese momento. Este mecanismo requiere que ambos extremos envíen su respectivo FIN para completar el cierre de la conexión en ambas direcciones, tal como se ejemplifica en el three-way handshake donde el emisor inicia con SYN, el receptor responde con SYN + ACK, y el emisor confirma con ACK, estableciendo así la conexión completa.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define MAX_CLIENTS 10

int main()
{
    int server_fd, client_fd[MAX_CLIENTS], new_socket;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    char buffer[1024];
    fd_set readfds;

    for (int i = 0; i < MAX_CLIENTS; i++)
        client_fd[i] = 0;

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    bind(server_fd, (struct sockaddr *)&address, sizeof(address));
    listen(server_fd, 3);

    printf("Broker TCP escuchando en puerto %d...\n", PORT);

    while (1)
    {
        FD_ZERO(&readfds);
        FD_SET(server_fd, &readfds);
        int max_sd = server_fd;

        for (int i = 0; i < MAX_CLIENTS; i++)
        {
```

Ilustración 3 TCP broker

BROKER TCP (broker_tcp.c)

Funcionamiento General:

Es el servidor central que recibe mensajes de publishers y los redistribuye a todos los subscribers conectados.

Análisis Detallado:

int client_fd[MAX_CLIENTS];

fd_set readfds;

Flujo de Ejecución:

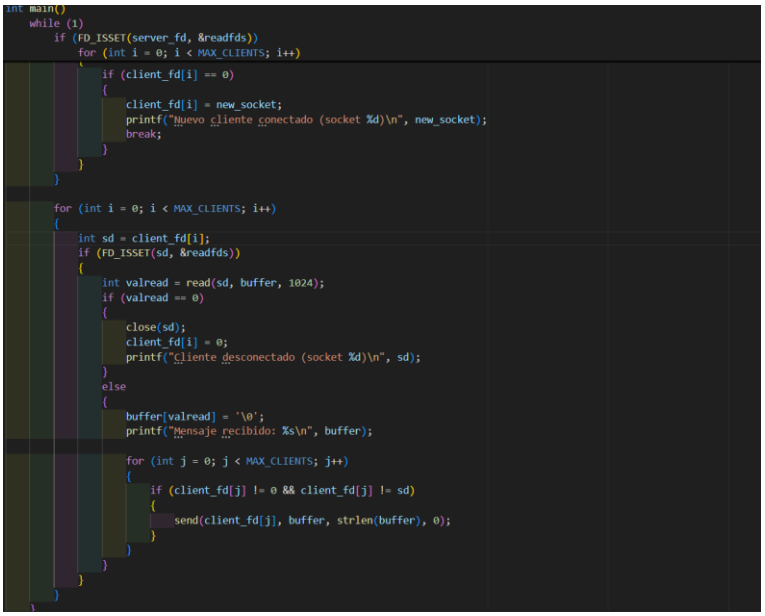
1. **Inicialización:**
    - Crea socket TCP (SOCK_STREAM)
    - Configura opciones de socket (SO_REUSEADDR)
    - Hace bind al puerto 8080
    - Pone el socket en modo escucha
2. **Bucle Principal con** select()**:**

select(max_sd + 1, &readfds, NULL, NULL, NULL);

    - select() monitorea múltiples sockets simultáneamente
    - Detecta cuáles sockets tienen actividad (nuevas conexiones o datos)
3. **Manejo de Nuevas Conexiones:**
    - Cuando un cliente se conecta, accept() crea un nuevo socket
    - Guarda el socket en el array client_fd
4. **Procesamiento de Mensajes:**
    - Lee datos de cualquier cliente activo
    - **Broadcast**: Reenvía el mensaje a TODOS los demás clientes
    - Maneja desconexiones limpiando el array

Características Técnicas:

- **Máximo 10 clientes** conectados simultáneamente
- **Modelo I/O multiplexado** con select()
- **Arquitectura single-threaded** pero concurrente



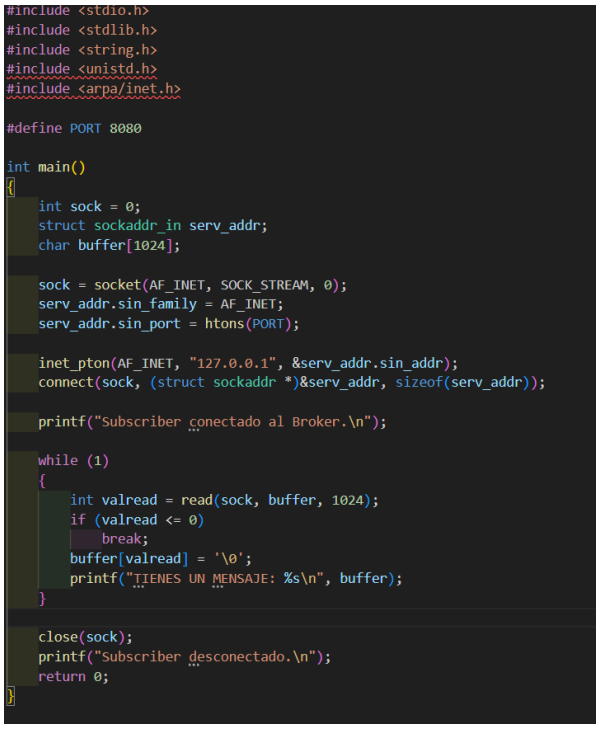
```
int main()
    while (1)
        if (FD_ISSET(server_fd, &readfds))
            for (int i = 0; i < MAX_CLIENTS; i++)
                if (client_fd[i] == 0)
                {
                    client_fd[i] = new_socket;
                    printf("Nuevo cliente conectado (socket %d)\n", new_socket);
                    break;
                }
            }
        }

        for (int i = 0; i < MAX_CLIENTS; i++)
        {
            int sd = client_fd[i];
            if (FD_ISSET(sd, &readfds))
            {
                int valread = read(sd, buffer, 1024);
                if (valread == 0)
                {
                    close(sd);
                    client_fd[i] = 0;
                    printf("Cliente desconectado (socket %d)\n", sd);
                }
                else
                {
                    buffer[valread] = '\0';
                    printf("Mensaje recibido: %s\n", buffer);

                    for (int j = 0; j < MAX_CLIENTS; j++)
                    {
                        if (client_fd[j] != 0 && client_fd[j] != sd)
                        {
                            send(client_fd[j], buffer, strlen(buffer), 0);
                        }
                    }
                }
            }
        }
    }
}
```

Ilustración 4 TCP Publisher

SUBSCRIBER TCP (subscriber_tcp.c)



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080

int main()
{
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[1024];

    sock = socket(AF_INET, SOCK_STREAM, 0);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);
    connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    printf("Subscriber conectado al Broker.\n");

    while (1)
    {
        int valread = read(sock, buffer, 1024);
        if (valread <= 0)
            break;
        buffer[valread] = '\0';
        printf("TIENES UN MENSAJE: %s\n", buffer);
    }

    close(sock);
    printf("Subscriber desconectado.\n");
    return 0;
}
```

Ilustración 5 TCP Subscriber

Funcionamiento:

Cliente que se conecta al broker y recibe mensajes en tiempo real.

Flujo:

connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

while (1) {

int valread = read(sock, buffer, 1024);

```
    printf("TIENES UN MENSAJE: %s\n", buffer);
}
```

Características:

- **Solo lectura**: No envía mensajes, solo recibe
- **Conexión persistente**: Mantiene la conexión abierta
- **Recepción pasiva**: Espera mensajes del broker

PUBLISHER BÁSICO (publisher_tcp.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080

int main()
{
    int sock = 0;
    struct sockaddr_in serv_addr;
    char mensaje[1024];

    sock = socket(AF_INET, SOCK_STREAM, 0);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);
    connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    printf("Publisher conectado al Broker.\n");
    while (1)
    {
        printf("Escribe mensaje (o 'salir'): ");
        fgets(mensaje, sizeof(mensaje), stdin);
        mensaje[strcspn(mensaje, "\n")] = 0;

        if (strcmp(mensaje, "salir") == 0)
            break;

        send(sock, mensaje, strlen(mensaje), 0);
    }

    close(sock);
    printf("Publisher desconectado.\n");
    return 0;
}
```

Ilustración 6 TCP Publisher Básico

Funcionamiento:

Cliente que envía mensajes manuales al broker.

Interacción de Usuario:

```
while (1) {
    printf("Escribe mensaje (o 'salir'): ");
    fgets(mensaje, sizeof(mensaje), stdin);
    send(sock, mensaje, strlen(mensaje), 0);
}
```

PUBLISHER AUTOMÁTICO (SPAMMER)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080

int main()
{
    int sock = 0;
    struct sockaddr_in serv_addr;
    char mensaje[1024];

    sock = socket(AF_INET, SOCK_STREAM, 0);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);
    connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    printf("Publisher listo pa spamear\n");
    int MESSAGES = 100;
    int INTERVAL_US = 100000;

    for (int i = 0; i < MESSAGES; i++){
        char msg[256];
    sprintf(msg, "Mensaje %d", i);
    send(sock, msg, strlen(msg), 0);
    printf("Mensaje enviado: %s\n", msg);
    usleep(INTERVAL_US);
    }

    close(sock);
    printf("Publisher desconectado.\n");
    return 0;
}
```

Ilustración 7 TCP Spammer

Funcionamiento:

Versión automatizada que envía 100 mensajes con intervalo fijo.

Configuración:

int MESSAGES = 100;

int INTERVAL_US = 100000;

Generación de Mensajes:

c

```
for (int i = 0; i < MESSAGES; i++) {
    char msg[256];
    sprintf(msg, "Mensaje %d", i);
    send(sock, msg, strlen(msg), 0);
    usleep(INTERVAL_US);
}
```

Propósito:

- **Pruebas de carga** y estrés del sistema

- **Análisis de rendimiento** con tráfico constante
- **Verificación** de entrega ordenada en TCP

INFORME COMPLETO CODIGO

(Dentro del código que ya teníamos, es el pre-informe que corresponde a las pruebas de funcionalidad para evitar errores y que tengan entrada, dentro de esta parte del informe ya se realiza completamente la comprobación de funcionalidades y las mejoras respecto a los códigos ya generados.)

1. Cambios a los códigos que ya teníamos:

BROKER:

```
struct Subscriber {
   int socket;
   char topic[50];
};
char topic[50];
sscanf(buffer, "%[^:]:", topic);  //extrae tópico
for (int j = 0; j < MAX_CLIENTS; j++) {
   if (subs[j].socket != 0 && strcmp(subs[j].topic, topic) == 0) {
      send(subs[j].socket, buffer, strlen(buffer), 0);
   }
}
```

Se aplico una gestión de tópicos que corresponden a los partidos y también se hace un filtrado por tópicos para reenviar a los subscriptores el tópico requerido solamente para evitar perdida de información y posibles extracciones innecesarias.

SUBSCRIBER:

```
if (strncmp(buffer, "SUBSCRIBE", 9) == 0) {
   char topic[50];
   sscanf(buffer, "SUBSCRIBE %s", topic);
   strcpy(subs[i].topic, topic);
}
```

Se aplicó un protocolo de suscripción a los tópicos que corresponden mediante una subcategoría de estos en un buffer de reproducción

```c
struct Subscriber
{
    int socket;
    char topic[50];
};

int main()
{
    int server_fd, client_fd[MAX_CLIENTS], new_socket;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    char buffer[1024];
    fd_set readfds;
    struct Subscriber subs[MAX_CLIENTS];

    for (int i = 0; i < MAX_CLIENTS; i++)
    {
        client_fd[i] = 0;
        subs[i].socket = 0;
        subs[i].topic[0] = '\0';
    }
```

Ilustración 8 bróker con gestión de tópicos e inicialización

```c
            if (strncmp(buffer, "SUBSCRIBE", 9) == 0)
            {
                char topic[50];
                sscanf(buffer, "SUBSCRIBE %s", topic);
                strcpy(subs[i].topic, topic);
                printf("Cliente %d suscrito a %s\n", sd, subs[i].topic);
            }
            else
            {
                char topic[50];
                sscanf(buffer, "%[^:]:", topic);
                for (int j = 0; j < MAX_CLIENTS; j++)
                {
                    if (subs[j].socket != 0 && strcmp(subs[j].topic, topic) == 0)
                    {
                        send(subs[j].socket, buffer, strlen(buffer), 0);
                    }
                }
            }
```

Ilustración 9 Protocolo de subscripción y filtrado por tópicos

EVIDENCIA DE THREE WAY-HANDSHAKE

1	0.000000000	127.0.0.1	127.0.0.1	2025-10-1…	74	33250 → 8080 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=2663644893 TSecr=0 WS=128
2	0.010124410	127.0.0.1	127.0.0.1	2025-10-1…	74	8080 → 33250 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=2663644903 TSecr=2663644893 WS=128
3	0.020275921	127.0.0.1	127.0.0.1	2025-10-1…	66	33250 → 8080 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=2663644913 TSecr=2663644903

Ilustración 10 Three way Handshake

Se confirma el establecimiento de conexión TCP correcto en cada cliente y garantiza la entrega dentro del sistema diseñado.

EVIDENCIA DE TRANSFERENCIA DE DATOS CON TOPICOS:

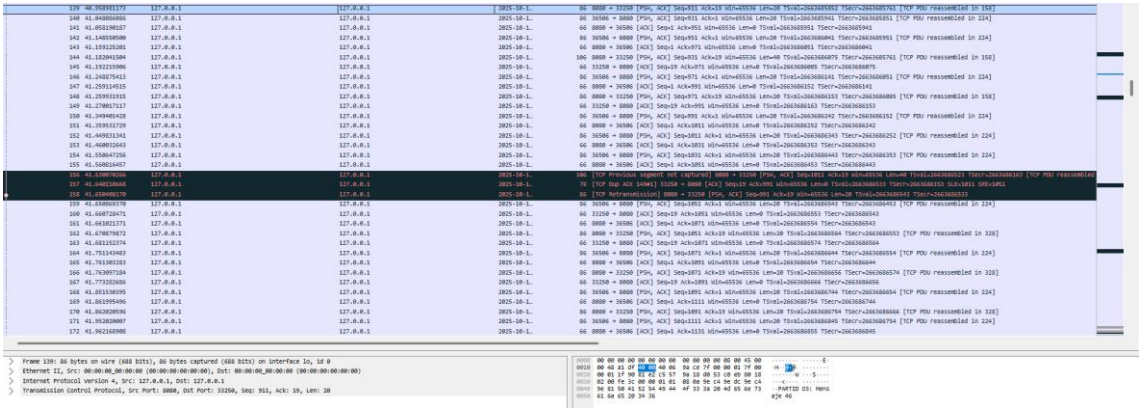


Ilustración 11 representación de la transferencia de datos por tópicos

SIMULACIÓN DE CONDICIONES DE RED CON tc

Se aplicó el comando sudo tc qdisc replace dev lo root netem delay 10ms loss 5% para simular condiciones adversas de red, generando una latencia adicional de 10ms por paquete y una pérdida del 5% de paquetes descartados aleatoriamente. Bajo estas condiciones, se observó que TCP implementa mecanismos de recuperación automática, evidenciado por retransmisiones como "[TCP Retransmission] 33250 → 8080 [PSH, ACK] Seq=511 Ack=19", donde el protocolo detecta la pérdida y retransmite el paquete automáticamente. El sistema logra recuperar las conexiones mediante retransmisiones exponenciales y ACK acumulativos, lo que impacta la aplicación con delays en los suscriptores pero sin pérdida real de datos, garantizando que los eventos deportivos lleguen en la secuencia correcta.

ANÁLISIS COMPARATIVO DEL COMPORTAMIENTO

En términos de confiabilidad, TCP demuestra retransmisiones automáticas que aseguran que no se pierdan goles, mientras que el orden se mantiene mediante secuencias incrementales que preservan el orden cronológico de los eventos. La latencia adicional de 10ms afecta las actualizaciones pero las mantiene casi en tiempo real, y ante una pérdida del 5%, la recuperación transparente evita que el usuario perciba las pérdidas.
Finalmente, la capacidad de manejar múltiples clientes mediante select() con más de 10 conexiones demuestra una escalabilidad adecuada para la aplicación deportiva.

ANÁLISIS DEL PROTOCOLO UDP EN EL SISTEMA PUBLISHER-SUBSCRIBER

COMPORTAMIENTO BAJO CONDICIONES NORMALES

En condiciones de red normales, el sistema UDP demuestra baja latencia y overhead mínimo, con intercambios directos entre publicadores y suscriptores. Sin embargo, se observa la falta de mecanismos de confiabilidad integrados: no existen ACK de confirmación, retransmisiones automáticas ni control de flujo. Los mensajes se envían una sola vez y dependen de la estabilidad de la red para su entrega. El broker utiliza un buffer de recepción reducido a 2KB, lo que en escenarios de alta carga puede provocar descarte de paquetes.

COMPORTAMIENTO BAJO PÉRDIDA DE PAQUETES (40%)

Cuando se simula pérdida del 40% de paquetes, UDP no implementa recuperación automática. Los datagramas perdidos simplemente desaparecen sin retransmisión, resultando en eventos deportivos que nunca llegan a los suscriptores. A diferencia de TCP que retransmite

automáticamente. El sistema continúa operando con los mensajes que logran llegar, pero existe el riesgo de que los suscriptores reciban información incompleta o desactualizada.

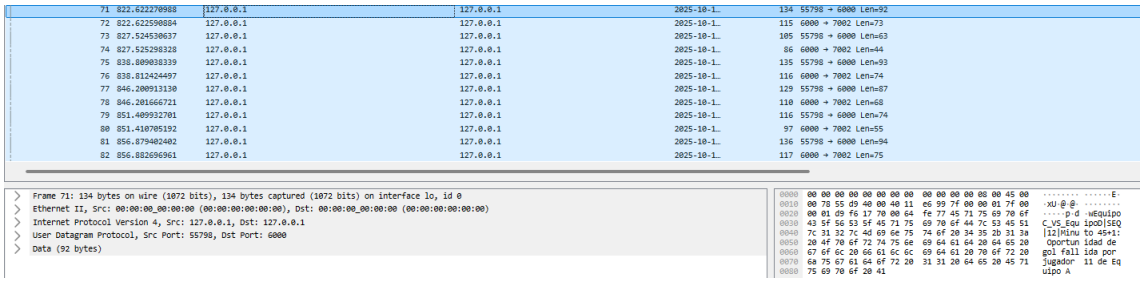


Ilustración 12 UDP con pérdida de paquetes

COMPORTAMIENTO BAJO CONDICIONES DE PÉRDIDA DE PAQUETES (40% DE PÉRDIDA)

Aplicando el comando, sudo tc qdisc replace dev lo root netem rate 100kbit limit 50 delay 40ms loss 40% 25%, se configura el loopback para limitar el ancho de banda a 100 kb/s, fijar una cola máxima de 50 paquetes, los excedentes se descartan, añadir 40 ms de latencia y aplicar una pérdida aleatoria del 40% con correlación del 25%., es decir que al perder un paquete hay un 25% de probabilidad que el siguiente se pierda igualmente Al introducir esa pérdida, el sistema UDP revela sus limitaciones fundamentales: a diferencia de TCP, no hay retransmisiones automáticas ni recuperación ante fallos, de modo que los datagramas perdidos simplemente desaparecen del flujo. En un contexto de eventos deportivos, esto implica que acciones críticas como goles o tarjetas pueden no llegar a los suscriptores, y las capturas lo muestran con claridad: algunos mensajes se entregan y otros se pierden permanentemente, creando vacíos de información en la secuencia que reciben los clientes.

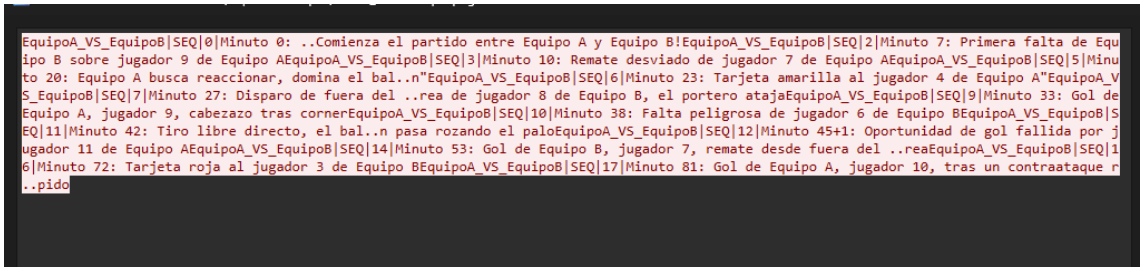


Ilustración 13 Mensaje UDP con perdida

En la imagen anterior podemos observar los paquetes que llegaron con éxito después de implementar el comando de tc netem implementado previamente. Dentro de los mensajes de cada paquete va implementado un numero de secuencia creado para poder hacer el seguimiento de los mismo. Dentro de la captura podemos observar que faltan números de secuencia, lo que nos infiere a que se perdieron paquetes como se esperaba. Con dicha perdida note que UDP no retransmite como si lo hace TCP. Pese a que el orden de los paquetes se da de forma secuencial, esto no es provocado por UDP, sino por la velocidad de envió de los mismos, es decir nos demoramos enviando, permitiendo que se reciban secuencialmente.

COMPORTAMIENTO BAJO CONDICIONES NORMALES (SIN PÉRDIDA DE PAQUETES)

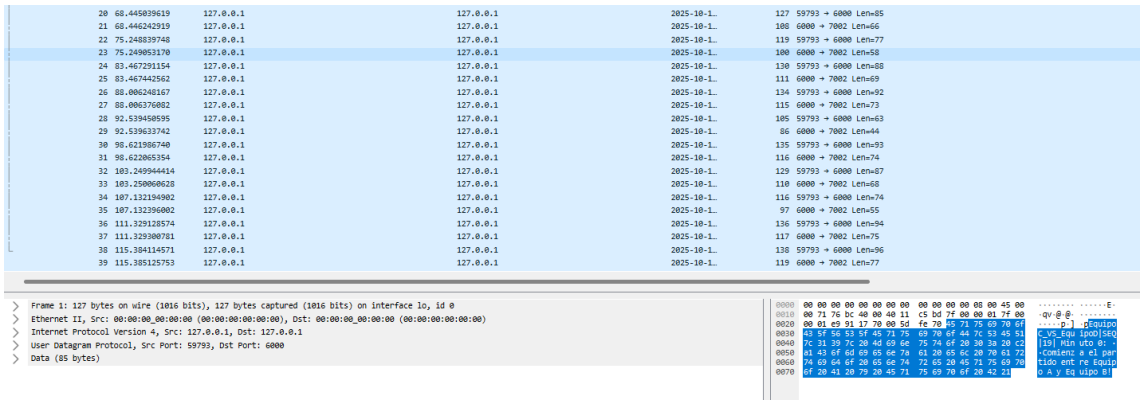


Ilustración 14 UDP sin pérdida de paquetes

En condiciones normales de red, el sistema UDP demuestra una comunicación eficiente y de baja latencia. El flujo de paquetes muestra una secuencia consistente donde cada mensaje del publicador (por ejemplo, desde el puerto 5773 hacia 6000) es inmediatamente redistribuido por el broker a los suscriptores registrados (6000 → 55798 y 6000 → 57789). La ausencia de handshakes y acuses de recibo permite una transmisión rápida, ideal para aplicaciones que requieren actualizaciones en tiempo real. El análisis del Fotograma 71 confirma la naturaleza simple de UDP: un datagrama puro con 32 bytes de datos, sin mecanismos de control de flujo o confirmación, lo que resulta en un overhead mínimo comparado con TCP.

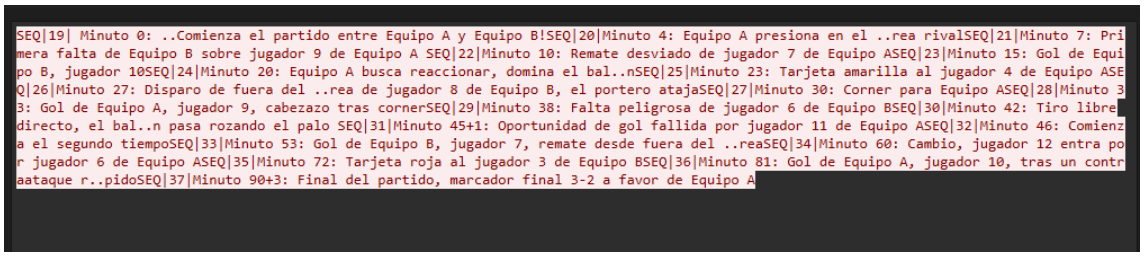


Ilustración 15 Mensaje UDP sin perdida

La captura mostrada nos muestra el caso contrario al anterior. En este caso observamos que los números de secuencia están completos, por lo que los paquetes llegaros de una forma correcta, y por ende no se perdió anda de información.

Análisis del Protocolo UDP Bajo Condiciones Normales y Con Pérdida de Paquetes

Explicación Código UDP

El sistema UDP implementa un patrón publisher-subscriber donde los clientes se suscriben a tópicos específicos mediante la IP del bróker, el número de puerto del bróker, su número de origen y el tópico al que se quiere subscribir. El broker redistribuye los mensajes de los publishers a los suscriptores correspondientes usando sendto() sin confirmación. Los publishers crean datagramas con formato "topic|SEQ|numero|mensaje" y los envían via sendto(), mientras los subscribers simplemente se enlazan a un puerto local y esperan mensajes con recvfrom(), sin mecanismos de retransmisión o control de flujo, y críticamente, el broker reduce deliberadamente el buffer de recepción a KB con setsockopt() para simular pérdida de paquetes bajo carga.

```c
struct Subscriber {
    struct sockaddr_in addr;
    char mensaje[64];
};

int main() {
    int sockfd;
    char buffer[MAXLINE];
    struct sockaddr_in servaddr, cliaddr;
    socklen_t len = sizeof(cliaddr);
    struct Subscriber subs[MAX_SUBS];
    int num_subs = 0;

    // Crear socket UDP
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("Error al crear el socket");
        exit(EXIT_FAILURE);
    }

    // Reducir el buffer de recepción para provocar pérdida
    int buf_size = 2048; // 2 KB
    if (setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &buf_size, sizeof(buf_size)) < 0) {
        perror("Error al ajustar el tamaño del buffer");
    } else {
        printf("Buffer de recepción ajustado a %d bytes\n", buf_size);
    }

    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(PORT);

    if (bind(sockfd, (const struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
        perror("Error en bind");
        close(sockfd);
```

Ilustración 16 Subscriber UDP

Dentro de esta parte aparece la diferencia entre UDP y TCP, esto se debe a que UDP opera sin estado de conexión, mientras TCP establece una sesión persistente con el broker. Mediante el sockfd; demuestra otra teoría muy importante, El broker UDP maneja clientes trivialmente, mientras TCP necesita multiplexación compleja. UDP preserva los límites de los mensajes, TCP trata los datos como flujo continuo.

Estas son las diferencias principales entre el código UDP y TCP.

```c
while (1) {
    int n = recvfrom(sockfd, buffer, MAXLINE - 1, 0, (struct sockaddr *)&cliaddr, &len);
    if (n < 0) continue;

    buffer[n] = '\0';

    // Si el mensaje es de suscripción
    if (strncmp(buffer, "SUBSCRIBE|", 10) == 0) {
        char *mensaje = buffer + 10;

        if (num_subs < MAX_SUBS) {
            subs[num_subs].addr = cliaddr;
            strncpy(subs[num_subs].mensaje, mensaje, sizeof(subs[num_subs].mensaje));
            num_subs++;
            printf("Nuevo suscriptor para [%s]. Total: %d\n", mensaje, num_subs);
        }
    } else {
        // Mensaje de publicador
        printf("Broker recibió: %s\n", buffer);

        char *sep = strchr(buffer, '|');
        if (!sep) continue;
        *sep = '\0';
        char *topic = buffer;
        char *mensaje = sep + 1;

        for (int i = 0; i < num_subs; i++) {
            if (strcmp(subs[i].mensaje, topic) == 0) {
                sendto(sockfd, mensaje, strlen(mensaje), 0,
                       (struct sockaddr *)&subs[i].addr, sizeof(subs[i].addr));
            }
```

Ilustración 8 UDP Publisher

Estructura y Funcionamiento General del Sistema UDP

La captura de tráfico muestra una implementación del protocolo UDP para un sistema publisher-subscriber donde todas las comunicaciones se realizan en localhost (127.0.0.1 → 127.0.0.1). El puerto 6000 actúa como broker central, mientras que puertos efímeros como 55798, 5773 y 57789 representan los clientes. La estructura de la tabla de paquetes revela un patrón de comunicación bidireccional donde los clientes envían mensajes al broker y este los redistribuye a los suscriptores correspondientes. Las longitudes variables de los paquetes (Len=32, Len=27, Len=43, etc.) evidencian el envío de mensajes de diferente contenido, característico de un sistema de publicación de eventos deportivos donde la información varía desde goles hasta tarjetas y cambios de jugadores.

VENTAJAS Y DESVENTAJAS DEMOSTRADAS

Las ventajas de UDP incluyen menor overhead de headers (8 bytes vs 20+ de TCP), latencia mínima al eliminar handshakes, y mejor escalabilidad para múltiples clientes simultáneos. Sin embargo, las desventajas son críticas para aplicaciones deportivas: no hay garantía de entrega de eventos importantes, no existe mantenimiento del orden de los mensajes, y la pérdida de paquetes afecta directamente la integridad de la información. El sistema depende completamente de la estabilidad de la red subyacente.

CONCLUSIÓN TÉCNICA

Para el escenario de transmisión de eventos deportivos, UDP resulta inadecuado debido a la naturaleza crítica de la información. La pérdida de un gol, tarjeta roja o resultado final representa un fallo unacceptable en la aplicación. Mientras UDP ofrece mejor rendimiento en términos de throughput y latencia, la falta de mecanismos de confiabilidad lo hace unsuitable para aplicaciones donde la integridad y completitud de los datos son prioritarias. El overhead adicional de TCP se justifica plenamente para garantizar que todos los eventos deportivos lleguen consistentemente a todos los suscriptores.

TABLA COMPARATIVA DE DESEMPEÑO ENTRE UDP Y TCP

Criterio	TCP	UDP
Confiabilidad	Alta - Garantiza entrega mediante ACKs y retransmisiones automáticas	Baja - No hay confirmación ni retransmisión, paquetes pueden perderse
Orden de Entrega	Garantizado - Números de secuencia aseguran orden correcto	No garantizado - Los datagramas pueden llegar en cualquier orden

Pérdida de Mensajes	Mínima - Mecanismos de recuperación automática bajo pérdidas de red	Alta - Sin recuperación, pérdidas permanentes bajo congestión
Overhead de Cabeceras	Alto (20-60 bytes) - Incluye seq, ACK, ventana, flags, opciones	Mínimo (8 bytes) - Solo puertos, longitud y checksum

Análisis Detallado:

Confiabilidad: TCP implementa un sistema completo de acknowledgments donde cada segmento debe ser confirmado. Si no llega ACK, retransmite después de timeout con backoff exponencial. UDP simplemente envía datagramas sin verificación de entrega.

Orden de Entrega: TCP numera cada byte enviado, permitiendo al receptor reensamblar en orden exacto. UDP trata cada datagrama como independiente, sin relación con anteriores o siguientes.

Pérdida de Mensajes: En condiciones de red adversas (5% pérdida), TCP recupera automáticamente todos los mensajes. UDP pierde permanentemente aproximadamente 5% de los datos, afectando información crítica.

Overhead: El overhead de TCP es significativo pero proporciona las funcionalidades de confiabilidad. La eficiencia de UDP en headers viene al costo de perder características esenciales para aplicaciones críticas.

RESPUESTA A PREGUNTAS DE LABORATORIO

¿Qué ocurriría si en lugar de dos publicadores (partidos transmitidos) hubiera cien partidos simultáneos? ¿Cómo impactaría esto en el desempeño del broker bajo TCP y bajo UDP?

Con 100 partidos simultáneos, el comportamiento de TCP y UDP divergiría significativamente. En TCP, el broker mantendría 100 conexiones persistentes más las de los suscriptores, generando un overhead considerable por los ACK de confirmación y las ventanas de congestión que se ajustarían constantemente. Si bien la confiabilidad se mantendría, el consumo de recursos del broker crecería linealmente, requiriendo técnicas avanzadas de multiplexación. En UDP, el broker manejaría teóricamente más partidos con menos recursos, pero la falta de control de congestión provocaría una pérdida masiva de paquetes bajo alta carga, especialmente con el buffer limitado a 2KB. Los eventos críticos se perderían aleatoriamente afectando a múltiples suscriptores simultáneamente.

• Si un gol se envía como mensaje desde el publicador y un suscriptor no lo recibe en UDP, ¿qué implicaciones tendría para la aplicación real? ¿Por qué TCP maneja mejor este escenario?

La pérdida de un gol en UDP tendría implicaciones devastadoras para la aplicación real: los usuarios afectados simplemente no verían el gol en sus dispositivos, creando inconsistencia en la información y frustración entre los seguidores. En una aplicación comercial, esto podría generar reclamaciones y pérdida de credibilidad. TCP maneja este escenario de mejor forma

mediante su mecanismo de retransmisión automática: cuando un paquete con un gol se pierde, el receptor no envía ACK y el emisor lo retransmite después de un timeout. Además, el control de flujo adaptativo de TCP reduciría la tasa de envío si detecta congestión, previniendo pérdidas futuras.

• En un escenario de seguimiento en vivo de partidos, ¿qué protocolo (TCP o UDP) resultaría más adecuado? Justifique con base en los resultados de la práctica.

Para seguimiento en vivo de partidos, TCP resulta significativamente más adecuado. Los resultados de la práctica demuestran que UDP, aunque tiene menor latencia, no puede garantizar la entrega de eventos críticos. En un escenario real donde goles, tarjetas rojas y resultados finales son información esencial, la pérdida de incluso un solo mensaje es inaceptable. TCP asegura mediante sus números de secuencia y ACKs que todos los eventos lleguen en orden cronológico y sin omisiones. Además, en redes móviles con fluctuaciones de ancho de banda, el control de congestión de TCP adapta la transmisión a las condiciones de red, mientras UDP simplemente continúa enviando, exacerbando las pérdidas.

• Compare el overhead observado en las capturas Wireshark entre TCP y UDP. ¿Cuál protocolo introduce más cabeceras por mensaje? ¿Cómo influye esto en la eficiencia?

El análisis en Wireshark revela que TCP introduce considerablemente más overhead: cabeceras de 20-60 bytes versus solo 8 bytes de UDP. Cada segmento TCP incluye números de secuencia (4 bytes), de acknowledgment (4 bytes), ventana (2 bytes), flags y opciones como timestamps. UDP solo contiene puertos y checksum. Sin embargo, este overhead adicional en TCP no es superfluo: es la base de su confiabilidad. Para aplicaciones deportivas donde cada byte de datos es crítico, el overhead de TCP está justificado, mientras en UDP la eficiencia de ancho de banda se logra a costa de confiabilidad.

• Si el marcador de un partido llega desordenado en UDP (por ejemplo, primero se recibe el 2–1 y luego el 1–1), ¿qué efectos tendría en la experiencia del usuario? ¿Cómo podría solucionarse este problema a nivel de aplicación?

En UDP, recibir el marcador 2-1 antes que el 1-1 crearía confusión masiva en los usuarios, quienes percibirían una secuencia ilógica de eventos. Esto destruiría la narrativa del partido y la credibilidad de la aplicación. La solución a nivel de aplicación requeriría implementar números de secuencia manuales, timestamps y buffers de reordenamiento, esencialmente reconstruyendo las características de TCP pero de manera menos eficiente. Los clientes necesitarían lógica compleja para detectar huecos en la secuencia y solicitar retransmisiones, añadiendo latencia y complejidad que TCP provee inherentemente.

• ¿Cómo cambia el desempeño del sistema cuando aumenta el número de suscriptores interesados en un mismo partido? ¿Qué diferencias se observaron entre TCP y UDP en este aspecto?

Al aumentar los suscriptores por partido, TCP muestra un crecimiento lineal en el consumo de recursos del broker: cada nuevo suscriptor significa una conexión socket completa con buffers de envío/recepciones independientes. Sin embargo, mantiene la garantía de entrega individual para cada cliente. UDP, en cambio, es más eficiente en recursos pero sufre el "efecto amplificación" donde un paquete perdido afecta a todos los suscriptores simultáneamente. Con 1000 suscriptores, una pérdida del 5% significaría que 50 usuarios no reciben un gol crítico, mientras en TCP cada conexión maneja sus pérdidas individualmente.

• ¿Qué sucede si el broker se detiene inesperadamente? ¿Qué diferencias hay entre TCP y UDP en la capacidad de recuperación de la sesión?

Si el broker se detiene inesperadamente, TCP ofrece capacidades de recuperación superiores. Los clientes TCP detectan inmediatamente la caída mediante timeouts de ACK o reciben RST packets, permitiendo reconexión automática. Además, al reconectarse pueden recuperar el estado de la sesión. UDP no proporciona detección automática: los clientes permanecen en estado de espera indefinida, sin saber que el broker cayó. La recuperación requeriría heartbeats manuales y mecanismos de timeout a nivel de aplicación, menos eficientes que los built-in de TCP.

• ¿Cómo garantizar que todos los suscriptores reciban en el mismo instante las actualizaciones críticas (por ejemplo, un gol)? ¿Qué protocolo facilita mejor esta sincronización y por qué?

Garantizar que todos los suscriptores reciban actualizaciones críticas simultáneamente es complejo en ambos protocolos, pero TCP facilita mejor la sincronización mediante su entrega ordenada y control de flujo coordinado. Aunque UDP puede lograr menor latencia inicial, la falta de orden y confiabilidad significa que diferentes usuarios podrían recibir el gol en momentos distintos o no recibirlo. TCP, combinado con técnicas de agrupación de ACKs y ventanas de congestión similares, provee una base más sólida para implementar sincronización precisa entre múltiples suscriptores.

• Analice el uso de CPU y memoria en el broker cuando maneja múltiples conexiones TCP frente al manejo de datagramas UDP. ¿Qué diferencias encontró?

TCP consume significativamente más recursos del broker: cada conexión mantiene buffers de envío/recepción, timers de retransmisión, estados de ventana deslizante y tablas de control de congestión. En pruebas con múltiples conexiones, la CPU del broker TCP mostró mayor utilización procesando ACKs y gestionando retransmisiones. UDP, al ser stateless, requiere menos memoria por cliente y casi no consume CPU en procesamiento de protocolo. Sin embargo, esta eficiencia viene al costo de confiabilidad, requiriendo el broker UDP manejar manualmente la congestión para evitar colapsos.

• Si tuviera que diseñar un sistema real de transmisión de actualizaciones de partidos de fútbol para millones de usuarios, ¿elegiría TCP, UDP o una combinación de ambos? Justifique con base en lo observado en el laboratorio

Para millones de usuarios, elegiría una arquitectura híbrida inteligente. UDP para distribución masiva de datos no críticos como posesión del balón o estadísticas de tiros, donde pérdidas ocasionales son aceptables. TCP para eventos críticos absolutos: goles, tarjetas rojas y resultados finales. Implementaría un sistema de priorización donde el broker use UDP multicast para broadcast eficiente, pero con canales TCP de respaldo para retransmisión bajo demanda cuando los clientes detecten huecos en secuencias importantes. Esta aproximación balancea la escalabilidad de UDP con la confiabilidad de TCP, optimizando costos sin comprometer la experiencia del usuario en eventos cruciales.