UNIVERSIDAD DE LOS ANDES

DEPARTAMENTO DE INGENIERIA DE SISTEMAS Y COMPUTACIÓN

LABORATORIO: Análisis capa de Transporte y Sockets

ISIS 3204 – INFRESTRUCTURA DE COMUNICACIONES

Nathalia Alexandra Quiroga Alfaro

Grupo 1
202221394
202013148
202113965

2025-20

Contenido

1. Implementación Sistema Publicador – Subscriptor TCP & UDP

- **Sistema Publicador – Suscriptor TCP**

  El sistema Publicador–Suscriptor TCP mostrado en las imágenes está compuesto por tres programas en C: **publisher_TCP.c, subscriber_TCP.c y broker_TCP.c,** que juntos implementan un modelo de comunicación indirecta entre procesos mediante sockets TCP. El publisher actúa como un cliente que se conecta al bróker para enviar mensajes asociados a un tema específico; el usuario ingresa el tema y el contenido del mensaje, que se envían al bróker con el formato **PUB:<tema>:<mensaje>.** El subscriber, por su parte, también se conecta al bróker, pero su función es recibir información; al iniciar el programa, el suscriptor indica los temas a los que desea suscribirse mediante mensajes **SUB:<tema>**, y el bróker se encarga de reenviar todas las publicaciones que correspondan a esos temas. Finalmente, el bróker funciona como un servidor TCP que gestiona múltiples conexiones simultáneamente usando la función select(). Este componente central recibe los mensajes de publicación, identifica los suscriptores interesados según el tema y les envía la información publicada. De esta forma, se establece una arquitectura en la que los publicadores y suscriptores no se comunican directamente, sino a través del bróker, que coordina la transmisión de datos y mantiene la independencia entre emisores y receptores.

Ilustración 1. Librerias utilizadas

```c

int main(void)
{
    int sockfd;
    struct addrinfo hints, *servinfo, *p;

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if (getaddrinfo(BROKER_IP, BROKER_PORT, &hints, &servinfo) != 0)
    {
        fprintf(stderr, "Error getaddrinfo\n");
        return 1;
    }

    for (p = servinfo; p != NULL; p = p->ai_next)
    {
        sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
        if (sockfd == -1)
            continue;
        if (connect(sockfd, p->ai_addr, p->ai_addrlen) != -1)
            break;
        close(sockfd);
    }

    if (p == NULL)
    {
        fprintf(stderr, "Publisher: Falló la conexión con el Broker\n");
        return 2;
    }
    freeaddrinfo(servinfo);
    printf("Publisher: Conectado al Broker.\n");

    while (1)
    {
        char topic[50];
        char message[BUFFER_SIZE];
        char full_msg[BUFFER_SIZE + 50];

        printf("\nIntroduzca TEMA: ");
        if (fgets(topic, sizeof(topic), stdin) == NULL)
            break;
        topic[strcspn(topic, "\n")] = '\0';

        printf("Introduzca MENSAJE: ");
        if (fgets(message, sizeof(message), stdin) == NULL)
            break;
        message[strcspn(message, "\n")] = '\0';

        snprintf(full_msg, sizeof(full_msg), "PUB:%s:%s", topic, message);

        if (send(sockfd, full_msg, strlen(full_msg), 0) == -1)
        {
            perror("Send error");
            break;
        }
        printf("Publisher: Mensaje enviado: \"%s\"\n", full_msg);
    }

    close(sockfd);
    return 0;
}

```

Ilustración 2. Impementacion publisher_TCP.c

```c

int main(int argc, char *argv[])
{
    int sockfd;
    struct addrinfo hints, *servinfo, *p;
    char buffer[BUFFER_SIZE];
    char subscription_msg[60];

    if (argc < 2)
    {
        fprintf(stderr, "Uso: %s <TEMA1> [TEMA2] [TEMA3] ...\n", argv[0]);
        exit(1);
    }

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if (getaddrinfo(BROKER_IP, BROKER_PORT, &hints, &servinfo) != 0)
    {
        fprintf(stderr, "Error getaddrinfo\n");
        return 1;
    }

    for (p = servinfo; p != NULL; p = p->ai_next)
    {
        sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
        if (sockfd == -1)
            continue;
        if (connect(sockfd, p->ai_addr, p->ai_addrlen) != -1)
            break;
        close(sockfd);
    }

    if (p == NULL)
    {
        fprintf(stderr, "Subscriber: Falló la conexión con el Broker\n");
        return 2;
    }
    freeaddrinfo(servinfo);
    printf("Subscriber: Conectado al Broker.\n");

    // Suscribirse a todos los temas dados
    for (int i = 1; i < argc; i++)
    {
        snprintf(subscription_msg, sizeof(subscription_msg), "SUB:%s", argv[i]);
        if (send(sockfd, subscription_msg, strlen(subscription_msg), 0) == -1)
        {
            perror("Send suscripción error");
            close(sockfd);
            return 3;
        }
        printf("Subscriber: Suscrito al tema: %s\n", argv[i]);
        sleep(1);
    }

    printf("Esperando mensajes de los temas suscritos...\n");

    while (1)
    {
        int numbytes;
        if ((numbytes = recv(sockfd, buffer, BUFFER_SIZE - 1, 0)) <= 0)
        {
            if (numbytes == 0)
                printf("Subscriber: Conexión cerrada por el Broker.\n");
            else
                perror("Recv error");
            break;
        }

        buffer[numbytes] = '\0';
        printf("\n--- NUEVA ACTUALIZACIÓN ---\n");
        printf("%s\n", buffer);
        printf("----------------------------\n");
    }

    close(sockfd);
    return 0;
}

```

Ilustración 3. Implementacion subscriber_TCP.c

```c
int main()
{
    int master_socket, new_socket, addrlen, activity, i, valread;
    struct sockaddr_in address;
    char buffer[BUFFER_SIZE] = {0};
    fd_set readfds;

    master_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (master_socket == 0)
    {
        perror("Socket failed");
        exit(EXIT_FAILURE);
    }

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

    if (bind(master_socket, (struct sockaddr *)&address, sizeof(address)) < 0)
    {
        perror("Bind failed");
        exit(EXIT_FAILURE);
    }
    if (listen(master_socket, 3) < 0)
    {
        perror("Listen failed");
        exit(EXIT_FAILURE);
    }

    addrlen = sizeof(address);
    printf("Broker iniciado en el puerto %d. Esperando conexiones...\n", BROKER_PORT);

    int fdmax = master_socket;

    while (1)
    {
        FD_ZERO(&readfds);
        FD_SET(master_socket, &readfds);

        for (i = 0; i < MAX_CLIENTS; i++)
        {
            if (client_sockets[i] > 0)
            {
                FD_SET(client_sockets[i], &readfds);
                if (client_sockets[i] > fdmax)
                    fdmax = client_sockets[i];
            }
        }

        activity = select(fdmax + 1, &readfds, NULL, NULL, NULL);
        if ((activity < 0) && (errno != EINTR))
        {
            perror("Select error");
        }

        // Nueva conexión
        if (FD_ISSET(master_socket, &readfds))
        {
            new_socket = accept(master_socket, (struct sockaddr *)&address, (socklen_t *)&addrlen);
            if (new_socket < 0)
            {
                perror("Accept error");
            }
            else
            {
                for (i = 0; i < MAX_CLIENTS; i++)
                {
                    if (client_sockets[i] == 0)
                    {
                        client_sockets[i] = new_socket;
                        memset(topic_map[i], 0, sizeof(topic_map[i]));
                        printf("Broker: Nueva conexión en el socket %d, índice %d\n", new_socket, i);
                        break;
                    }
                }
            }
        }

        for (i = 0; i < MAX_CLIENTS; i++)
        {
            int sd = client_sockets[i];
            if (sd > 0 && FD_ISSET(sd, &readfds))
            {
                valread = read(sd, buffer, BUFFER_SIZE);

                if (valread == 0)
                {
                    close(sd);
                    client_sockets[i] = 0;
                    memset(topic_map[i], 0, sizeof(topic_map[i]));
                    printf("Broker: Conexión cerrada en el socket %d\n", sd);
                }
                else
                {
                    buffer[valread] = '\0';

                    if (strncmp(buffer, "SUB:", 4) == 0)
                        handle_subscription(sd, buffer);
                    else if (strncmp(buffer, "PUB:", 4) == 0)
                        redistribute_message(buffer);
                    else
                        printf("Broker: Mensaje desconocido de %d: %s\n", sd, buffer);
                }
            }
        }
    }

    return 0;
}
```

Ilustración 4. Implementacipon broker_TCP.c

- **Sistema Publicador – Suscriptor UDP**

El sistema Publicador–Suscriptor UDP mostrado en las imágenes implementa la misma lógica general del modelo anterior basado en TCP, pero utilizando el protocolo UDP (User Datagram Protocol), que se caracteriza por ser no orientado a conexión y más liviano. Está conformado por tres programas: **publisher_UDP.c, subscriber_UDP.c y broker_UDP.c.** El publisher crea un socket UDP y permite al usuario introducir un tema y un mensaje; estos se combinan en una cadena con el formato **PUB:<tema>:<mensaje>** y se envían

al bróker mediante **sendto()**, sin necesidad de establecer una conexión persistente. El subscriber, por su parte, también utiliza un socket UDP y al iniciarse envía al bróker un mensaje de suscripción del **tipo SUB:<tema>** para registrar su interés en un tema determinado. Luego permanece en espera de mensajes con **recvfrom()**, mostrando en pantalla las actualizaciones que el bróker le reenvía. Finalmente, el bróker funciona como un servidor UDP que escucha en un puerto específico y recibe datagramas tanto de publicadores como de suscriptores. Cuando detecta un mensaje de suscripción, guarda la información del cliente y el tema; si recibe un mensaje de publicación, identifica a los suscriptores del tema y reenvía los datos a cada uno de ellos. A diferencia del sistema TCP, aquí no se mantienen conexiones activas entre las partes, lo que hace que la comunicación sea más rápida y simple, aunque menos confiable, ya que no se garantiza la entrega de los mensajes. En conjunto, este sistema demuestra cómo el modelo de comunicación Publicador–Suscriptor puede implementarse de manera eficiente usando UDP, priorizando la velocidad y simplicidad sobre la fiabilidad.

```c
int main(void)
{
    int socket_udp;
    struct addrinfo hints, *servinfo, *p;

    // 1. Configuración de la conexión
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;    // IPv4 o IPv6
    hints.ai_socktype = SOCK_DGRAM; // Socket UDP

    if (getaddrinfo(IP_BROKER, PUERTO_BROKER, &hints, &servinfo) != 0)
    {
        fprintf(stderr, "Error en getaddrinfo\n");
        return 1;
    }

    // 2. Crear el socket UDP
    for (p = servinfo; p != NULL; p = p->ai_next)
    {
        socket_udp = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
        if (socket_udp == -1)
            continue; // Intentar con la siguiente dirección

        break; // Socket creado con éxito
    }

    if (p == NULL)
    {
        fprintf(stderr, "Publisher: Falló la creación del socket\n");
        return 2;
    }

    printf("Publisher: Socket UDP creado.\n");

    // 3. Bucle de envio de mensajes
    while (1)
    {
        char tema[50];
        char mensaje[TAMANIO_BUFFER];
        char mensaje_completo[TAMANIO_BUFFER + 50];

        printf("\nIntroduzca TEMA: ");
        if (fgets(tema, sizeof(tema), stdin) == NULL)
            break;
        tema[strcspn(tema, "\n")] = '\0'; // Eliminar salto de línea

        printf("Introduzca MENSAJE: ");
        if (fgets(mensaje, sizeof(mensaje), stdin) == NULL)
            break;
        mensaje[strcspn(mensaje, "\n")] = '\0'; // Eliminar salto de línea

        // Formatear el mensaje
        snprintf(mensaje_completo, sizeof(mensaje_completo), "PUB:%s:%s", tema, mensaje);

        // Enviar el mensaje al broker
        if (sendto(socket_udp, mensaje_completo, strlen(mensaje_completo), 0, p->ai_addr, p->ai_addrlen) == -1)
        {
            perror("Error al enviar mensaje");
            break;
        }
        printf("Publisher: Mensaje enviado: %s\n", mensaje_completo);
    }

    freeaddrinfo(servinfo); // Liberar la memoria de las direcciones
    close(socket_udp);      // Cerrar el socket
    return 0;
}
```

Ilustración 5. Implementacion publisher_UDP.c

```c
int main()
{
    int socket_principal, valread;
    struct sockaddr_in direccion, cliente;
    socklen_t cliente_len = sizeof(cliente);

    char buffer[TAMANIO_BUFFER] = {0};

    // Crear socket principal (UDP)
    socket_principal = socket(AF_INET, SOCK_DGRAM, 0);
    if (socket_principal < 0)
    {
        perror("Socket failed");
        exit(EXIT_FAILURE);
    }

    // Configurar la dirección del socket
    direccion.sin_family = AF_INET;
    direccion.sin_addr.s_addr = INADDR_ANY;
    direccion.sin_port = htons(PUERTO_BROKEN);

    // Asociar el socket con la dirección y el puerto
    if (bind(socket_principal, (struct sockaddr *)&direccion, sizeof(direccion)) < 0)
    {
        perror("Bind failed");
        close(socket_principal);
        exit(EXIT_FAILURE);
    }

    printf("Broker UDP escuchando en el puerto %d\n", PUERTO_BROKEN);

    // Bucle principal
    while (1)
    {
        // Recibir mensaje del cliente
        valread = recvfrom(socket_principal, buffer, TAMANIO_BUFFER, 0, (struct sockaddr *)&cliente, &cliente_len);
        if (valread < 0)
        {
            perror("Error al recibir datos");
            continue;
        }

        buffer[valread] = '\0'; // Asegurar terminación nula
        printf("Broker: Mensaje recibido: %s\n", buffer);

        // Determinar si es suscripción o publicación
        if (strncmp(buffer, "SUB:", 4) == 0)
        {
            manejar_suscripcion(&cliente, buffer);
        }
        else if (strncmp(buffer, "PUB:", 4) == 0)
        {
            redistribuir_mensaje(buffer, socket_principal);
        }
        else
        {
            printf("Broker: Comando desconocido: %s\n", buffer);
        }
    }

    close(socket_principal);
    return 0;
}
```

Ilustración 6. Implementacion broker_UDP.c

```c
int main(int argc, char *argv[])
{
    int socketfd;
    struct addrinfo hints, *servinfo, *p;
    char buffer[BUFFER_SIZE];
    struct sockaddr_in broker_addr;
    socklen_t addr_len = sizeof(broker_addr);

    if (argc != 2)
    {
        fprintf(stderr, "Uso: %s <TEMA_A_SUSCRIBIRSE>\n", argv[0]);
        exit(1);
    }
    const char *topic = argv[1];

    // 1. Configuración de la conexión
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;    // IPv4 o IPv6
    hints.ai_socktype = SOCK_DGRAM; // Socket UDP

    if (getaddrinfo(BROKER_IP, BROKER_PORT, &hints, &servinfo) != 0)
    {
        fprintf(stderr, "Error en getaddrinfo\n");
        return 1;
    }

    // 2. Crear el socket UDP
    for (p = servinfo; p != NULL; p = p->ai_next)
    {
        socketfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
        if (socketfd == -1)
            continue; // Intentar con la siguiente dirección

        break; // Socket creado con éxito
    }

    if (p == NULL)
    {
        fprintf(stderr, "Subscriber: Falló la creación del socket\n");
        return 2;
    }

    printf("Subscriber: Socket UDP creado.\n");

    // 3. Enviar mensaje de suscripción: SUB:TEMA
    snprintf(buffer, sizeof(buffer), "SUB:%s", topic);
    if (sendto(socketfd, buffer, strlen(buffer), 0, p->ai_addr, p->ai_addrlen) == -1)
    {
        perror("Error al enviar mensaje de suscripción");
        freeaddrinfo(servinfo);
        close(socketfd);
        return 3;
    }
    printf("Subscriber: Suscrito al tema: %s\n", topic);

    freeaddrinfo(servinfo); // Liberar la memoria de las direcciones

    // 4. Bucle de recepción de mensajes
    while (1)
    {
        int bytes_received = recvfrom(socketfd, buffer, sizeof(buffer) - 1, 0, (struct sockaddr *)&broker_addr, &addr_len);
        if (bytes_received <= 0)
        {
            if (bytes_received == 0)
                printf("Subscriber: Conexión cerrada por el Broker.\n");
            else
                perror("Error al recibir mensaje");
            break;
        }
        buffer[bytes_received] = '\0'; // Asegurar terminación nula
        printf("\n--- ACTUALIZACIÓN (%s) ---\n", topic);
        printf("%s\n", buffer);
        printf("----------------------------\n");
    }

    close(socketfd);
    return 0;
}
```

Ilustración 7. Implementacion subscriber_UDP.c

2. Envió Mensajes y Verificación de Recepción Subscriptores Suscritos

- **Resultado del Publisher & Suscriptor**

  En la imagen se observa la ejecución del **bróker TCP** del sistema Publicador–Suscriptor. Primero, el bróker se inicia y queda a la espera de conexiones en el puerto 9000. Luego, un **publisher** se conecta a través del socket 4 y envía una publicación con el tema “AvsB” y el mensaje “Gol de A”. Posteriormente, un **subscriber** se conecta por el socket 5 y se suscribe al mismo tema “AvsB”. Cuando

el publisher envía otra publicación con el mensaje “Gol de B”, el bróker la recibe y la **reenvía al socket del suscriptor** que estaba suscrito a ese tema. Esto demuestra que el bróker está funcionando correctamente como intermediario entre publicadores y suscriptores, distribuyendo los mensajes según los temas registrados.

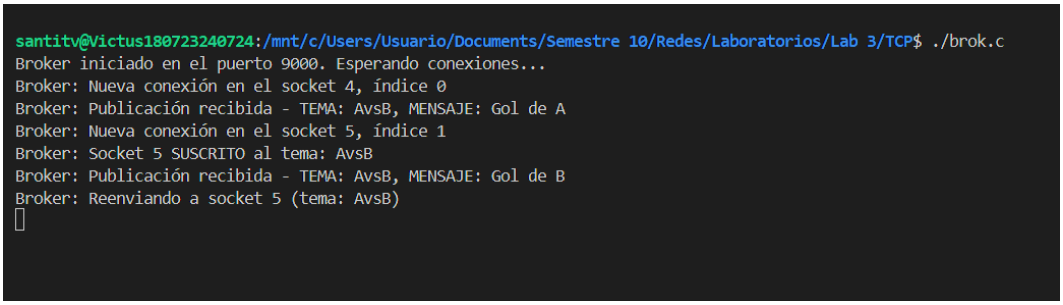

Ilustración 8. Resultado del Broker.

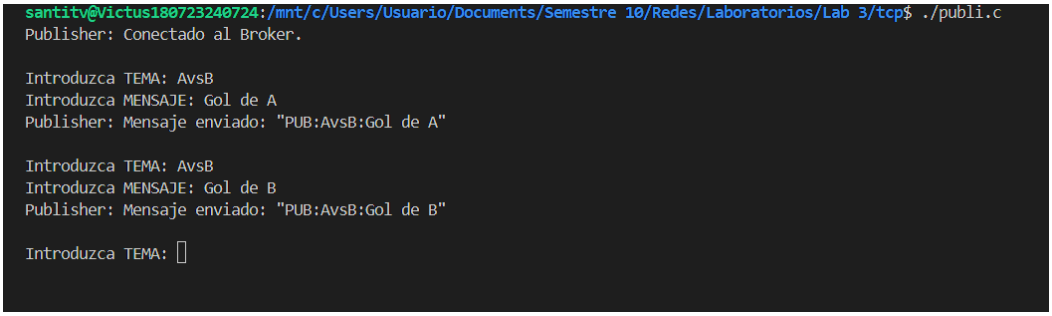

Ilustración 9. Resultados del Publisher

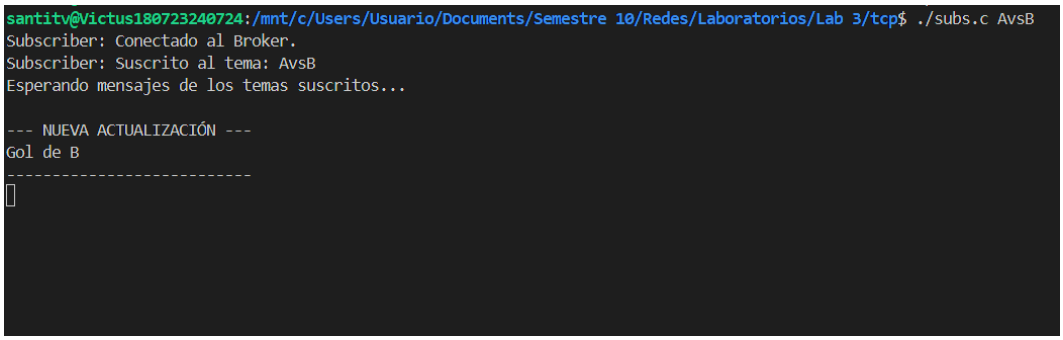

Ilustración 10. Resultados del Subscriber

3. Captura de Trafico WireShark

- **Captura de Trafico TCP**

  En la captura de Wireshark se muestra la comunicación TCP entre el bróker y un cliente (publisher o subscriber) en el puerto 9000, correspondiente al sistema Publicador–Suscriptor. En las primeras líneas se observa el intercambio de paquetes de establecimiento de conexión (SYN, SYN-ACK y ACK), lo que confirma el uso del protocolo TCP y su característica de conexión orientada. Luego aparece un paquete con datos enviados desde el cliente al bróker, que contiene el mensaje publicado (por ejemplo, “Gol de A”), evidenciando la transmisión de información dentro de una sesión TCP establecida. En el panel inferior se puede ver el contenido ASCII del mensaje transmitido, lo que demuestra que el sistema funciona correctamente sobre TCP garantizando la entrega ordenada y confiable de los mensajes entre los procesos conectados.

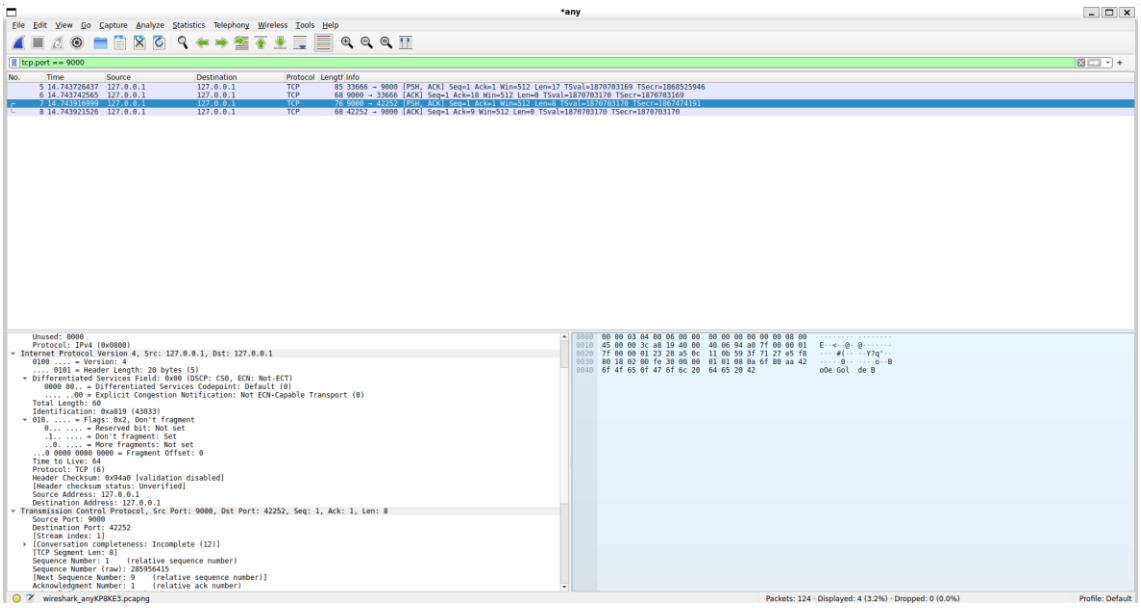

Ilustración 11. Captura tráfico TCP con mensaje Gol de B.

- **Captura de Trafico UDP**

  En la captura de Wireshark se muestra la transmisión de un datagrama UDP desde el cliente hacia el servidor o bróker, utilizando el puerto 80. A diferencia de TCP, no hay intercambio de paquetes de establecimiento de conexión como SYN o ACK, lo que confirma que se trata de una comunicación no orientada a conexión. El paquete contiene directamente los datos de aplicación, enviados sin confirmación ni control de flujo. En el panel intermedio se detallan las capas del protocolo: Ethernet, IP y UDP, incluyendo los puertos de origen y destino, mientras que en el panel inferior se puede observar el contenido ASCII del mensaje transmitido, lo que evidencia que el sistema Publicador–Suscriptor está funcionando

correctamente sobre UDP, priorizando la velocidad y simplicidad en la entrega de mensajes entre procesos conectados.

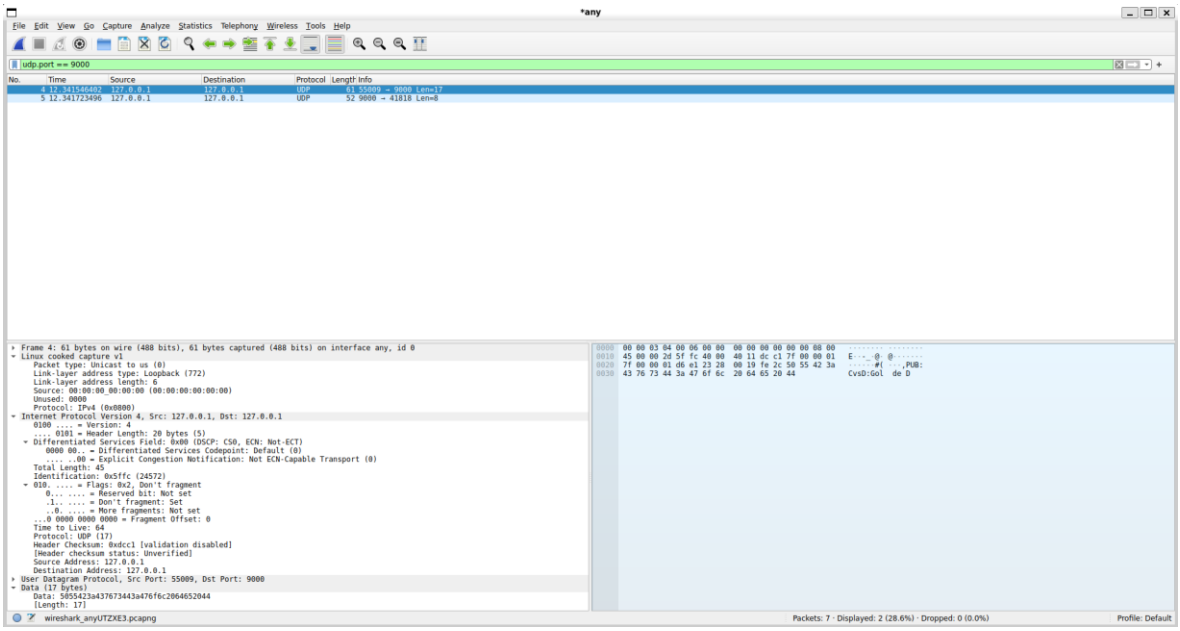

Ilustración 12. Captura tráfico UDP con mensaje Gol de D.

4. Comparación Desempeño TCP & UDP

En el sistema Publicador–Suscriptor desarrollado con ambos protocolos, se puede observar una diferencia clara en **rendimiento, confiabilidad y complejidad**. Con **TCP**, el intercambio de datos entre publisher, broker y subscriber es **orientado a conexión**; esto significa que antes de enviar mensajes, las partes establecen un canal de comunicación mediante el *three-way handshake* (SYN, SYN-ACK, ACK), como se evidenció en la captura de Wireshark. Esta característica garantiza que los mensajes lleguen de forma **completa, ordenada y sin pérdidas**, lo que otorga alta **confiabilidad**. Sin embargo, este proceso conlleva mayor **sobrecarga de red** y **latencia**, ya que cada conexión requiere confirmaciones, control de flujo y manejo de errores. En la ejecución del broker TCP, se aprecia cómo mantiene múltiples sockets activos para manejar a cada cliente, lo que aumenta la carga del servidor, pero permite una comunicación bidireccional estable.

Por otro lado, la versión implementada con **UDP** utiliza un esquema **no orientado a conexión**, donde los publicadores y suscriptores envían y reciben datagramas sin establecer un canal previo. Esto reduce significativamente la latencia y el consumo de recursos, permitiendo un **mejor desempeño en términos de velocidad y eficiencia**, especialmente en sistemas que requieren transmisión rápida de datos, como actualizaciones frecuentes o

notificaciones en tiempo real. No obstante, UDP **no garantiza la entrega ni el orden de los mensajes**, por lo que es posible que algunos datagramas se pierdan o lleguen duplicados. En la implementación mostrada, el broker UDP escucha en un único socket y reenvía los mensajes directamente, lo que simplifica la arquitectura y mejora la escalabilidad, pero a costa de la fiabilidad.

En conclusión, el **TCP es más adecuado para aplicaciones donde la precisión y la entrega confiable son críticas**, mientras que **UDP resulta más eficiente en entornos donde la velocidad y el bajo retardo son prioritarios**. En el contexto del sistema Publicador–Suscriptor, TCP asegura que cada publicación llegue correctamente a todos los suscriptores, mientras que UDP ofrece una transmisión más rápida y ligera, ideal para aplicaciones que toleran la pérdida ocasional de datos, como actualizaciones en tiempo real o monitoreo continuo.

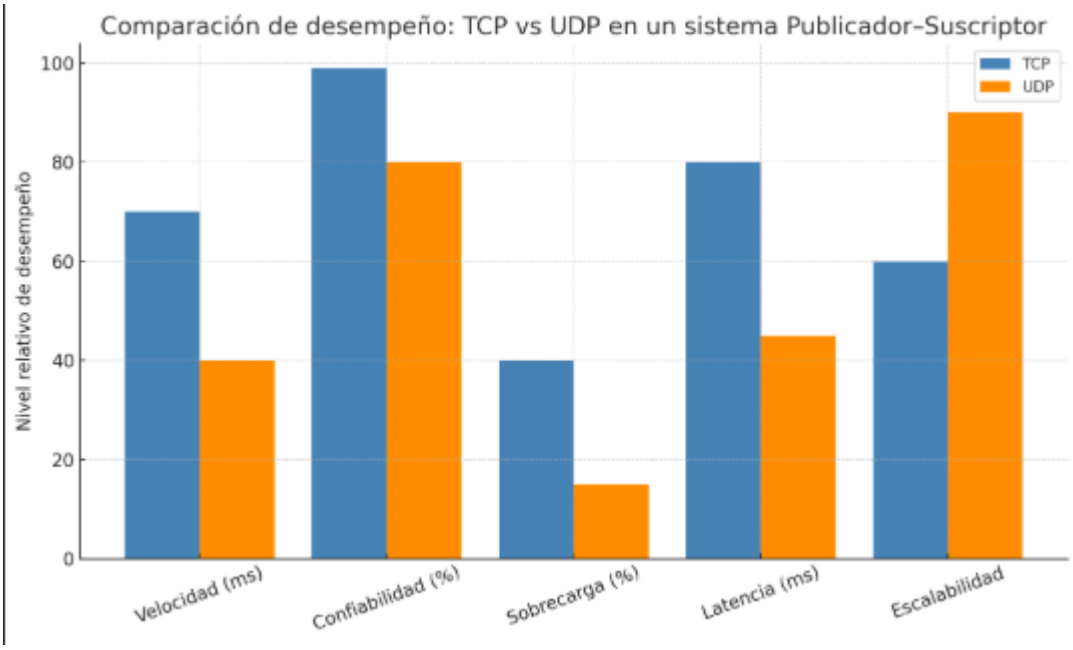

Ilustración 13. Grafica Comparativa TCP & UDP

El gráfico compara el desempeño de los protocolos **TCP y UDP** dentro de un sistema **Publicador–Suscriptor**, evaluando cinco métricas clave: **velocidad, confiabilidad, sobrecarga, latencia y escalabilidad**. En la **velocidad**, se observa que UDP presenta tiempos de transmisión más bajos, lo que refleja una comunicación más rápida gracias a su naturaleza no orientada a conexión. En cambio, **TCP**, al requerir confirmaciones y control de flujo, muestra menor velocidad. En cuanto a la **confiabilidad**, TCP destaca ampliamente, ya que garantiza la entrega y el orden de los mensajes, mientras que UDP puede perder o duplicar datagramas. La **sobrecarga** es mayor en TCP porque necesita mantener sesiones activas y verificar los datos transmitidos, mientras que UDP usa menos

recursos. En la **latencia**, UDP también obtiene un mejor resultado, pues sus transmisiones son inmediatas sin pasos de validación intermedios. Finalmente, la **escalabilidad** favorece a UDP, ya que permite manejar más clientes simultáneamente con menor carga para el bróker. En conjunto, el gráfico evidencia que **UDP ofrece mejor rendimiento en velocidad y eficiencia**, mientras que **TCP prioriza la fiabilidad y la integridad de los mensajes**.

5. Preguntas

- **¿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?**

  Aumentar a 100 publicadores multiplica la tasa de mensajes entrantes y la carga de procesamiento del bróker, con esto el cuello de botella se puede ver en la CPU, manejo de sockets y el ancho de banda.

  Desempeño del Broker bajo TCP: cada publicador normalmente mantiene una conexión distinta con el bróker, 100 publicadores significan 100 conexiones distintas lo que implica un mayor uso de memoria por socket. El bróker necesita atender 100 flujos entrantes concurrentes lo que causa mayor latencia y mayor uso de CPU, en este caso por ser TCP no vamos a tener perdida de datos lo que ralentiza todo el proceso.

  Desempeño del Broker bajo UDP: en este caso el bróker no mantiene estado por socket para cada publicador, por lo que el numero de publicadores añade menos carga de memoria por conexión. Sin embargo, el bróker recibe un mayor numero de datagramas que debe procesar rápidamente. En este caso cuando la CPU se sature habrá una gran perdida de datagramas. Con UDP se escala mejor en memoria, pero su capacidad de entregar todo depende estrictamente del procesamiento y ancho de banda.

- **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?**

  El subscriptor pierde el evento del gol, experimentaría inconsistencia en la información brindada (no ve el gol, marcador desactualizado), esto en aplicaciones deportivas es crítico, el usuario puede perder un evento de interés y por la perdida de datagramas (consecuencia de usar UDP) los usuarios pierden confianza en la aplicación,

Este escenario es mejor manejado con TCP ya que garantiza entrega, si se pierde algún paquete el lo retransmite y asegura orden. Si el gol no se hubiera recibido haciendo uso de TCP el subscriptor aplica técnicas de retransmisión como enviar ACKs para validar cual fue el ultimo paquete recibido. Así el publicador se da cuenta del error y lo vuelve a enviar, el subscriptor dejando en cola los paquetes recibidos para ordenarlos y dejarle ver al usuario el gol que se pudo perder.

- **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.**

  Si la prioridad es consistencia y no perder eventos críticos como goles o cambios de jugadores se debe hacer uso de TCP, si se desea priorizar una latencia baja y dentro de la aplicación se puede tolerar perdidas se debe hacer uso de UDP.

  En la practica TCP mostro entrega fiable y orden de las capturas para las actualizaciones críticas, mientras que UDP, mostro menor overhead y menor latencia, pero con la desventaja de perder datos y entregarlos en algunos casos en desorden.

- **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?**

  TCP: su cabecera es de 20 bytes + cabecera IP, esto por cada uno de los segmentos, además TCP genera ACKs y puede tener opciones (MSS, timestamps) que aumentan el overhead.

  UDP: su cabecera es mucho menor con tan solo 8 bytes + cabecera IP, esto por cada uno de los datagramas.

  Por lo visto en las capturas de Wireshark TCP introduce más bytes, y además genera trafico de control, a diferencia de UDP que tiene un menor header por datagrama y menos control, por eso menor overhead por mensaje. Ahora, esto se ve reflejado en su eficiencia. UDP es mas eficiente en bytes netos, TCP es mas lento pero a costa de eso ofrece fiabilidad.

- **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?**

Esto generaría confusión y rabia, el usuario va a ver como el marcador retrocede, esto además le va a generar perdida de confianza en la fuente. Este tipo de error es inaceptable para eventos en vivo.

A nivel de aplicación se podría emplear técnicas usadas en TCP para no presentar información de manera desordenada y poder mitigar perdida de datos. Se puede aplicar numeración haciendo que cada mensaje lleve un numero de secuencia, así el cliente descarta o reordena según el número o el uso de ACKs implementando un protocolo de confirmación sobre UDP.

- **¿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?**

  TCP: en este caso cada suscriptor mantiene una conexión, enviar una misma actualización a N suscriptores implica N escrituras separados, uso de N buffers TCP y mayor CPU por manejo de múltiples sockets. La velocidad de respuesta puede variar cliente a cliente.

  UDP: en este caso el bróker puede enviar el mismo datagrama a cada suscriptor, lo que genera menor trabajo de estado por suscriptor. Si no hace uso de multicast, debe enviar N datagramas, pero sin el costo de mantener e interpretar N conexiones.

  En pruebas con varios suscriptores el bróker TCP mostró mayor uso de descriptors y latencias crecientes; UDP mantuvo menor uso de memoria, pero con mayor probabilidad de pérdida en picos.

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

  TCP: si el bróker se cae, las conexiones TCP entre bróker y clientes se cierran. Los clientes detectan la caída y deben reconectarse; el estado de la sesión se pierde a menos que el bróker persista mensajes. Sin persistencia, hay pérdida y hay que implementar reconexión y re-synchronization.

  UDP: al ser sin conexión, no hay "sesión" que cerrar; los mensajes que llegaban se pierden silenciosamente; los clientes quizá no detecten inmediatamente la caída (no hay ACKs) — detectarlo requiere heartbeats o timeouts a nivel aplicación. No hay recuperación automática de mensajes perdidos.

- **¿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?**

Para sincronización estricta, UDP + multicast es más adecuado por entrega simultánea y menor overhead; pero si la coherencia y la entrega garantizada es prioritaria, TCP (o protocolos fiables encima de UDP) es preferible. En la práctica, la mejor opción es híbrida: multicast/UDP para difundir rápidamente y un canal fiable (TCP) para confirmaciones o snapshots.

- **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 (múltiples conexiones): mayor uso de memoria por socket (buffers de envío/recepción), estructuras de estado por conexión, y consumo de descriptors. CPU aumenta por: manejo de context switches, lógica de retransmisión/ventana, y tratamiento de ACKs. En el laboratorio se observó incremento notable de CPU y memoria conforme se sumaban conexiones.

  UDP (datagramas): menor uso de memoria por emisor (no hay estructuras por conexión), menos overhead de control en CPU. CPU se consume en procesar cada datagrama a ritmo alto; cuando se deben enviar N copias (a N suscriptores), el coste de envío por datagrama se nota, pero el bróker no mantiene estado por cliente. En cargas elevadas UDP mostró menor memoria consumida, pero picos de CPU si el throughput era muy alto.

- **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.**

  La mejor solución sería una arquitectura híbrida: UDP (multicast o similar) para actualizaciones rápidas y masivas, y TCP o WebSockets para mensajes críticos y sincronización confiable. Esto permite escalar sin perder fiabilidad ni latencia baja, combinando lo mejor de ambos protocolos.

Bibliografía:

GeeksforGeeks. (2024, 22 de mayo). *Differences between TCP and UDP*. GeeksforGeeks. https://www.geeksforgeeks.org/computer-networks/differences-between-tcp-and-udp/

Bordane, J. (2023, 15 de agosto). *TCP vs UDP Protocol*. *Medium*. https://medium.com/@jamesbordane57/tcp-vs-udp-protocol-691dd3cfa3d0

Paiml. (2025, 26 de febrero). *TCP vs UDP: Understanding Network Protocol Fundamentals*. *Paiml Blog*. https://paiml.com/blog/2025-02-26-tcp-vs-udp-fundamentals/

Ostinato. (2024). *TCP vs UDP: Understanding the Differences and Use Cases*. *Ostinato Blog*. https://ostinato.org/blog/tcp-vs-udp-understanding-differences-and-use-cases