Instituto Politécnico Nacional

Escuela Superior de Cómputo


Ingeniería en Sistemas Computacionales

Aplicaciones para comunicaciones en red

Academia “Sistemas Distribuidos”

Plan 20

Práctica 2:

# “**Sockets Orientados a Conexiones no Bloqueantes**”

2015090269

González González Armando Omar

Profesor: Ojeda Santillan Rodrigo

# Contenido

# Objetivo

Esta práctica tiene como propósito desarrollar y demostrar el uso eficiente de sockets no bloqueantes en aplicaciones de red mediante el lenguaje de programación C. Se enfoca en la implementación de servidores capaces de gestionar múltiples conexiones de clientes simultáneamente, sin afectar el rendimiento del servidor ni la capacidad de respuesta del sistema. Esta habilidad resulta fundamental en aplicaciones en tiempo real, como servidores de juegos en línea, plataformas de mensajería y sistemas de transacciones financieras. A lo largo de la práctica, el estudiante aprenderá a configurar sockets no bloqueantes y a administrar conexiones concurrentes, lo que es crucial para el desarrollo de aplicaciones de alto rendimiento.

# Introducción

En aplicaciones de red que requieren alta concurrencia y tiempos de respuesta rápidos, como los servidores web o los videojuegos en línea, es fundamental evitar que el servidor se detenga mientras espera la finalización de una operación de red, como la lectura o escritura de datos. El uso de sockets bloqueantes puede hacer que el hilo principal quede inactivo durante este tiempo de espera, lo que impacta negativamente en la latencia y el rendimiento del sistema.

Para solucionar este inconveniente, los sockets no bloqueantes permiten que el programa siga ejecutándose mientras las operaciones de red están en curso. En lugar de quedar detenido, el socket devuelve un error que indica que la operación debe intentarse nuevamente más adelante. Gracias a esto, un servidor puede atender a múltiples clientes al mismo tiempo sin interrupciones.

En esta práctica, se emplea el modelo cliente-servidor para demostrar cómo el uso de sockets no bloqueantes mejora la capacidad del servidor para gestionar varias conexiones de manera eficiente. Se hace uso de funciones como select(), que permite monitorear múltiples conexiones simultáneamente y determinar cuándo están listas para lectura o escritura.

# Desarrollo

## Cliente

El cliente realizará una conexión hacia el servidor utilizando un socket. El proceso es muy similar a lo que realizamos en la práctica 1, incluso cuando el lenguaje de programación sea distinto, muchos de los métodos tienen nombres similares.

El cliente crea un socket e intenta realizar una conexión hacia localhost (o 127.0.0.1) en el puerto 5555 que nosotros definimos.

Posteriormente procede a mandar un mensaje hacia el servidor, en este caso la cadena de caracteres “Hola desde el cliente”. Espera la respuesta del servidor y la imprime, para finalmente cerrar el socket y terminar el programa.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <sys/socket.h>
#define PORT 5555
#define BUFFER_SIZE 1024

int main() {
  int sock = 0;
  struct sockaddr_in serv_addr;
  char *hello = "Hola desde el cliente";
  char buffer[BUFFER_SIZE] = {0};

  // Crear socket
  if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("Socket creation error");
    return -1;
  }

  serv_addr.sin_family = AF_INET;
  serv_addr.sin_port = htons(PORT);

  // Convertir direcciones IPv4 e IPv6 de texto a binario
  if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
    perror("Invalid address/ Address not supported");
    return -1;
  }
```

```
    // Conectarse al servidor
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        perror("Connection Failed");
        return -1;
    }

    // Enviar mensaje al servidor
    send(sock, hello, strlen(hello), 0);
    printf("Mensaje enviado\n");

    // Leer respuesta del servidor
    read(sock, buffer, BUFFER_SIZE);
    printf("Respuesta del servidor: %s\n", buffer);

    // Cerrar el socket
    close(sock);

    return 0;
}
```

Bloque de código 1. Código del cliente.

## Servidor

El servidor en este caso se encarga de crear un socket para aceptar conexiones, en este caso el socket se configura en modo no bloqueante pasando la constante ***O_NONBLOCK***, también se configura el socket para recibir conexiones de entrada desde cualquier IP pero que la conexión llegue en específico al puerto 5555 de la computadora donde se ejecuta el servidor.

Procede a escuchar conexiones a través de ese socket (a lo mucho recibe hasta 30 sockets entrantes para conexión) y por medio de un ciclo ***while*** infinito acepta y maneja esas conexiones.

En ese ciclo el servidor espera actividad, si detecta alguna significa que es una nueva conexión entrante, por lo que crea un socket de conexión aceptando dicha conexión, e imprime los datos del origen de dicha conexión con su FD (file descriptor), su IP y el puerto de origen y agrega el socket entrante al arreglo de sockets.

Posteriormente maneja las entradas y salidas de los otros sockets: en un proceso donde primero revisa si el socket fue cerrado y lee el mensaje que mandó el socket, si no ha sido cerrado el socket de conexión entrante, reenvía el mensaje que el cliente mandó al propio cliente.

El programa termina cuando el usuario force la terminación de la ejecución del programa, o cuando ocurra alguna excepción que no esté manejada.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <fcntl.h>
#include <sys/select.h>
#define PORT 5555
#define BUFFER_SIZE 1024

int main() {
  int server_fd, new_socket, max_sd, sd, activity, valread;
  int client_socket[30] = {0};
  int max_clients = 30;
  int addrlen;
  struct sockaddr_in address;
  char buffer[BUFFER_SIZE];
  fd_set readfds;

  // Crear socket del servidor
  if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
    perror("socket failed");
    exit(EXIT_FAILURE);
  }

  // Configurar el socket del servidor como no bloqueante
  fcntl(server_fd, F_SETFL, O_NONBLOCK);

  // Configurar el tipo de socket
  address.sin_family = AF_INET;
  address.sin_addr.s_addr = INADDR_ANY;
  address.sin_port = htons(PORT);

  // Adjuntar el socket al puerto 5555
  if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
    perror("bind failed");
    close(server_fd);
    exit(EXIT_FAILURE);
  }

  // Escuchar en el socket
  if (listen(server_fd, 3) < 0) {
    perror("listen failed");
    close(server_fd);
    exit(EXIT_FAILURE);
  }
```

```
  printf("Escuchando en el puerto %d \n", PORT);

  // Bucle principal para aceptar y manejar conexiones
  while (1) {
    // Limpiar el conjunto de descriptores de socket
    FD_ZERO(&readfds);

    // Añadir el socket del servidor al conjunto de descriptores
    FD_SET(server_fd, &readfds);
    max_sd = server_fd;

    // Añadir los sockets de cliente al conjunto de descriptores
    for (int i = 0; i < max_clients; i++) {
      sd = client_socket[i];

      if (sd > 0)
        FD_SET(sd, &readfds);

      if (sd > max_sd)
        max_sd = sd;
    }

    // Esperar a que ocurra alguna actividad en uno de los sockets
    activity = select(max_sd + 1, &readfds, NULL, NULL, NULL);

    if (activity < 0) {
      perror("select error");
    }

    // Si hay una actividad en el socket del servidor, es una nueva conexión
    if (FD_ISSET(server_fd, &readfds)) {
      addrlen = sizeof(address);

      if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t
*)&addrlen)) < 0) {
        perror("accept failed");
        exit(EXIT_FAILURE);
      }

      printf("Nueva conexión, socket fd es %d, ip es : %s, puerto : %d\n", new_socket,
inet_ntoa(address.sin_addr), ntohs(address.sin_port));

      // Añadir el nuevo socket al array de sockets de cliente
      for (int i = 0; i < max_clients; i++) {
        if (client_socket[i] == 0) {
          client_socket[i] = new_socket;
          printf("Añadiendo a la lista de sockets como %d\n", i);
          break;
```

```
        }
      }
    }

    // Manejar IO en otros sockets
    for (int i = 0; i < max_clients; i++) {
      sd = client_socket[i];

      if (FD_ISSET(sd, &readfds)) {
        // Revisar si fue por cierre y leer el mensaje
        if ((valread = read(sd, buffer, BUFFER_SIZE)) == 0) {
          // Alguien se desconectó, obtener detalles e imprimir
          getpeername(sd, (struct sockaddr *)&address, (socklen_t *)&addrlen);
          printf("Host desconectado, ip %s, puerto %d\n", inet_ntoa(address.sin_addr),
ntohs(address.sin_port));

          // Cerrar el socket y marcarlo como 0 en la lista
          close(sd);
          client_socket[i] = 0;
        } else {
          // Poner terminador de cadena en el buffer y enviar mensaje de vuelta al
cliente
          buffer[valread] = '\0';
          send(sd, buffer, strlen(buffer), 0);
        }
      }
    }
  }

  return 0;
}
```

Bloque de Código 2. Código del servidor.


Figura 1. Ejecución del servidor y del cliente en dos terminales a la par.

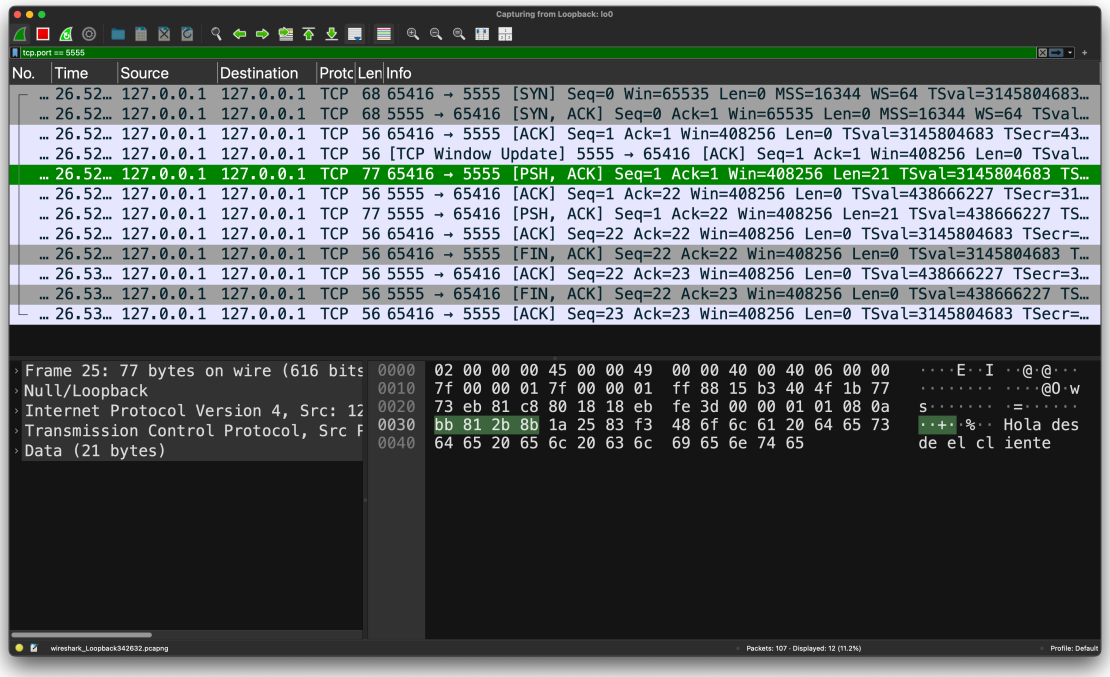


Figura 2. Captura en Wireshark del paquete enviado del cliente al servidor con el mensaje “Hola desde el cliente”.

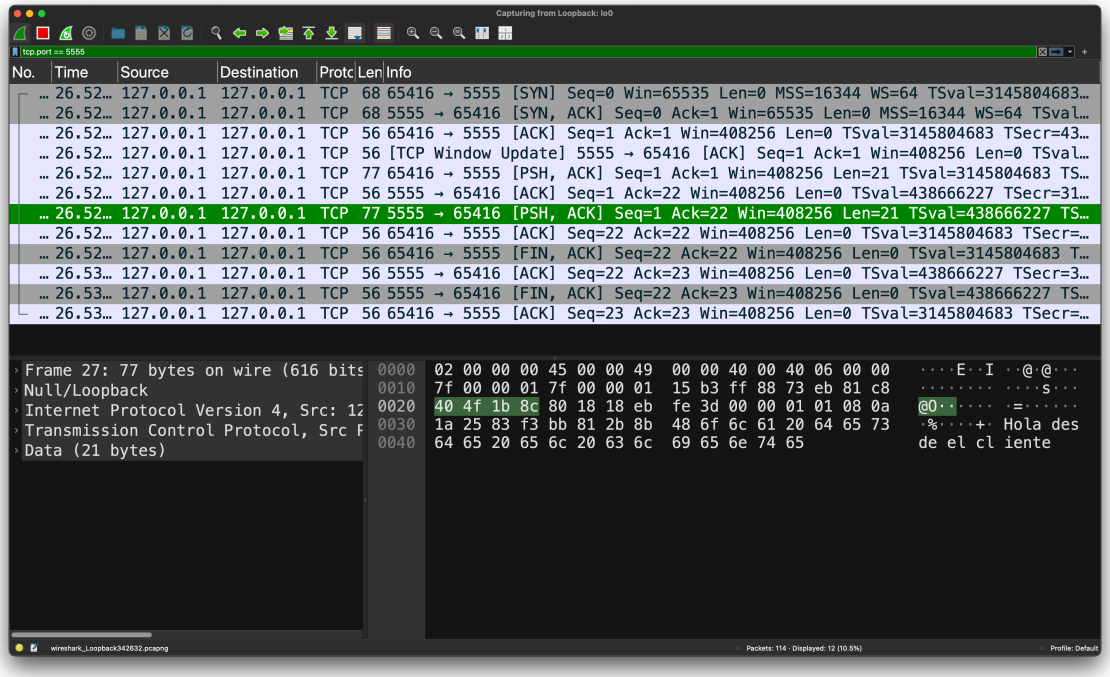


Figura 3. Captura en Wireshark del paquete enviado desde el servidor al cliente, con el mismo mensaje que mandó el cliente.


Figura 4. Dos ejecuciones adicionales donde se puede ver que el puerto de origen de la conexión ha cambiado.

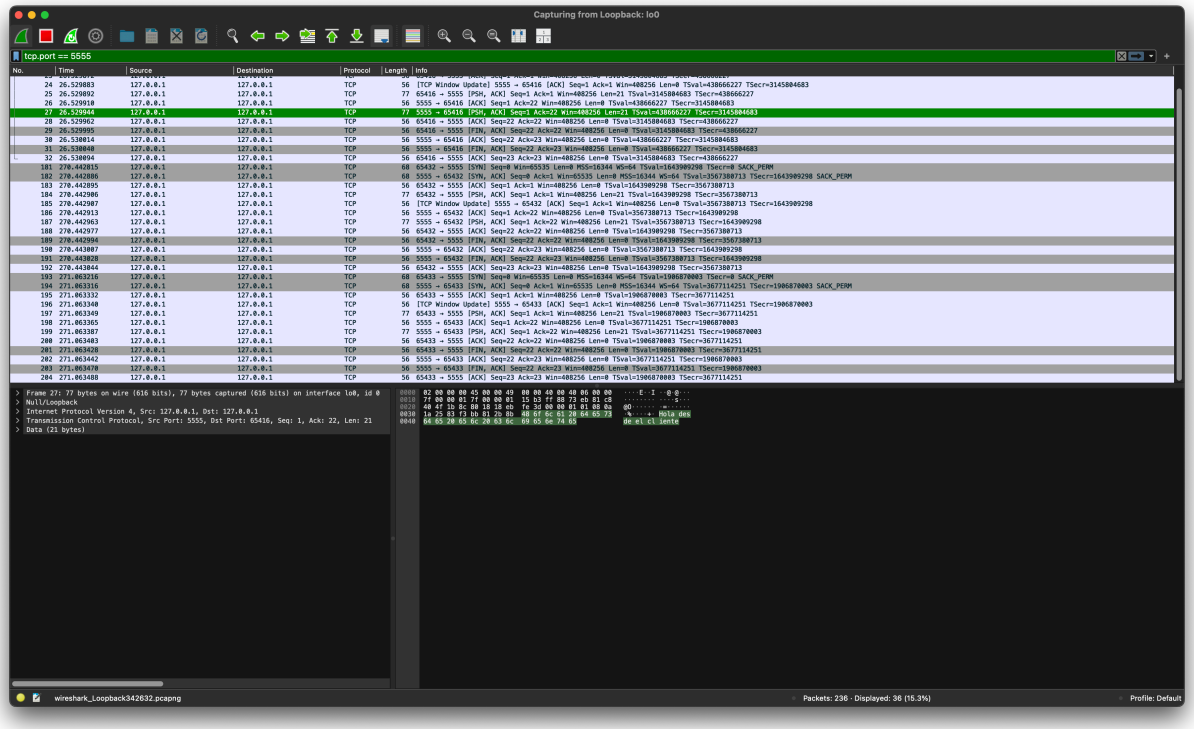


Figura 5. Captura en Wireshark de los paquetes de esas dos ejecuciones adicionales.


Figura 6. Ejecución del cliente cuando el servidor no está ejecutando.

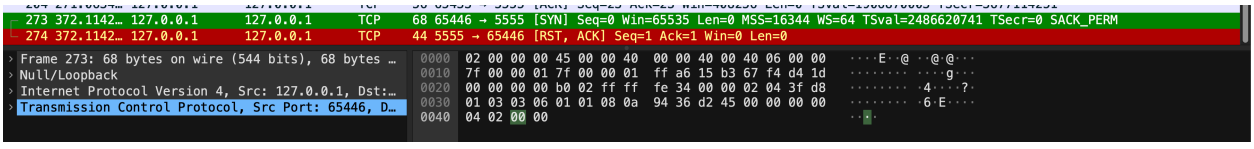


Figura 7. Captura en Wireshark del intento de envió de paquete al servidor inactivo.

# Conclusión

## González González Armando Omar

El usar sockets orientados a conexiones no bloqueantes, nos permiten realizar conexiones sin estar a la espera de que recibamos una conexión, esto permite que podamos atender múltiples peticiones, no necesariamente en paralelo, pero sí atenderlas y pasar a atender una siguiente conexión, como sería en el caso del código de la práctica, nos permite realizar esas conexiones múltiples y el server estaría encargado de escuchar y atender cada una de ellas.

# ¿Cómo podrían generar un negocio a través de lo visto en la práctica?

Los sockets orientados a conexiones no bloqueantes (usando métodos y funciones como **`select()`**, **`poll()`**, **`epoll()`** o **`kqueue()`**) permiten construir sistemas escalables de alta performance.

Al igual que en la práctica anterior, podríamos implementar un chat, pero en este caso que permita la conexión de múltiples usuarios para recepción y envío de mensajes, y que estos mensajes sean recibidos en tiempo real.

# Bibliografía


- **Begg, A.** *Sockets TCP/IP en C: Guía Práctica para Programadores*. Morgan Kaufmann, 2000.
- **Forouzan, Behrouz A.** *Comunicaciones de Datos y Redes*. McGraw-Hill, 2012.
- **Kurose, James F., y Keith W. Ross.** *Redes de Computadoras: Un Enfoque Descendente*. Pearson, 2017.