DPTO. DE INFORMÁTICA Y ANÁLISIS NUMÉRICO

UNIVERSIDAD DE CÓRDOBA

REDES – Práctica 2

Titulación: I.T. Informática Sistemas

PRÁCTICA 2

1. Fundamentos de la práctica 2

La programación de aplicaciones sobre TCP/IP se basa en el llamado modelo cliente-servidor. Básicamente, la idea consiste en que al indicar un intercambio de información, una de las partes debe "iniciar" el diálogo (cliente) mientras que la otra debe estar indefinidamente preparada a recibir peticiones de establecimiento de dicho diálogo (servidor). Cada vez que un usuario cliente desee entablar un diálogo, primero deberá contactar con el servidor, mandar una petición y posteriormente esperar la respuesta.

Los servidores pueden clasificarse atendiendo a si están diseñados para admitir múltiples conexiones simultáneas (servidores concurrentes), por oposición a los servidores que admiten una sola conexión por aplicación ejecutada (llamados iterativos). Evidentemente, el diseño de estos últimos será más sencillo que el de los primeros.

Otro concepto importante es la determinación del tipo de interacción entre el cliente y el servidor, que puede ser orientada a conexión o no orientada a conexión. Éstas corresponden, respectivamente, con los dos protocolos característicos de la capa de transporte: TCP y UDP. TCP (orientado a conexión) garantiza toda la "fiabilidad" requerida para que la transmisión esté libre de errores: verifica que todos los datos se reciben, automáticamente retransmite aquellos que no fueron recibidos, garantiza que no hay errores de transmisión y además, numera los datos para garantizar que se reciben en el mismo orden con el que fueron transmitidos. Igualmente, elimina los datos que por algún motivo aparecen repetidos, realiza un control de flujo para evitar que el emisor envíe más rápido de lo que el receptor puede consumir y, finalmente, informa a las aplicaciones (tanto cliente como al servidor) si los niveles inferiores de red no pueden entablar la conexión. Por el contrario, UDP (no orientada a conexión) no introduce ningún procedimiento que garantice la seguridad de los datos transmitidos, siendo en este caso responsabilidad de las aplicaciones la realización de los procedimientos necesarios para subsanar cualquier tipo de error.

En el desarrollo de aplicaciones sobre TCP/IP es imprescindible conocer como éstas pueden intercambiar información con los niveles inferiores; es decir, conocer la interfaz con los protocolos TCP o UDP. Esta interfaz es bastante análoga al procedimiento de entrada/salida ordinario en el sistema operativo UNIX que, como se sabe, está basado en la secuencia abrir-leer/escribir-cerrar. En particular, la interfaz es muy similar a los descriptores de fichero usados en las operaciones convencionales de entrada/salida en UNIX. Recuérdese que en las operaciones entrada/salida es necesario realizar la apertura del fichero (*open*) antes de que la aplicación pueda acceder a dicho fichero a través del ente abstracto "descriptor de fichero". En

la interacción de las aplicaciones con los protocolos TCP o UDP, es necesario que éstas obtengan antes el descriptor o "*socket*", y a partir de ese momento, dichas aplicaciones intercambiarán información con el nivel inferior a través del socket creado. Una vez creados, los sockets pueden ser usados por el servidor para esperar indefinidamente el establecimiento de una conexión (*sockets pasivos*) o, por el contrario, pueden ser usados por el cliente para iniciar la conexión (*sockets activos*).

1.1 ESTRUCTURA Y FUNCIONES ÚTILES EN LA INTERFAZ SOCKET

Para el desarrollo de aplicaciones, el sistema proporciona una serie de funciones y utilidades que permiten el manejo de los sockets. Puesto que muchas de las funciones y estructuras son iguales que las desarrolladas para los sockets UDP y éstas han sido explicadas en la práctica 1. En esta sección se detallaran solamente aquellas funciones nuevas.

1.1.1 Estructuras de la interfaz socket

Se parte de las estructuras sockaddr, sockaddr_in definidas en la práctica anterior, estudiaremos aquí otras estructuras interesantes.

Estructura hostent

La estructura hostent definida en el fichero /usr/include/netdb.h, contiene entre otras, la direccion IP del *host* en binario:

```
struct hostent {
        char *h_name;  /* nombre del host oficials */
        char  **h_aliases;  /* otros alias */
        int  h_addrtype;  /* tipo de dirección */
        int h_lenght;   /* longitud de la dirección en bytes */
        char **h_addr_list; /* lista de direcciones para el host */
}
#define h_addr h_addr_list[0]
```

Asociado con la estructura hostent está la función gethostbyname, que permite la conversión entre un nombre de host del tipo *www.uco.es* a su representación en binario en el campo *h_addr* de la estructura hostent.

Estructura servent

La estructura servent (también definida en el fichero netdb.h) contiene, entre otros, como campo el número del puerto con el que se desea comunicar:

```
struct servent {
        char *s_name;  /* nombre oficial del servicio */
        char  **s_aliases;  /* otros alias */
        int  s_port;   /* número del puerto para este servicio */
        char *s_proto;   /* protocolo a usar */
}
```

Con la estructura servent se relaciona la función getservbyname que permite a un cliente o servidor buscar el número oficial de puerto asociado a una aplicación estándar.

1.1.2 Funciones de la interfaz socket

TCP se caracteriza por tener un paso previo de establecimiento de la conexión, con lo que existen una serie de primitivas que no existían en el caso de UDP y que se estudiarán en este apartado.

Ambos, cliente y servidor, deben crear un socket mediante la función socket(), para poder comunicarse. El uso de esta función es igual que el descrito en la práctica 1 con la especificación de que se va a usar el protocolo SOCK_STREAM.

Otras funciones que no se han visto en la práctica1 y que se emplearán en TCP se detallan en este apartado.

Función listen()

Se llama desde el servidor, habilita el socket para que pueda recibir conexiones.

```
/* Se habilita el socket  para recibir conexiones */
int listen ( int sockfd, int backlog)
```

Esta función admite dos parámetros:

- *(1º argumento, sockfd),* es el descriptor del socket devuelto por la función socket() que será utilizado para recibir conexiones.
- *(2º argumento, backlog),* es el número máximo de conexiones en la cola de entrada de conexiones. Las conexiones entrantes quedan en estado de espera en esta cola hasta que se aceptan.

Función accept()

Se utiliza en el servidor, con un socket habilitado para recibir conexiones (listen()). Esta función retorna un nuevo descriptor de socket al recibir la conexión del cliente en el puerto configurado. La llamada a accept() no retornará hasta que se produce una conexión o es interrumpida por una señal.

```
/* Se queda a la espera hasta que lleguen  conexiones */
int accept ( int sockfd, struct sockaddr *addr, socklen_t *addrlen)
```

Esta función admite tres parámetros:

- *(1º argumento, sockfd),* es el descriptor del socket habilitado para recibir conexiones.
- *(2º argumento, addr),* puntero a una estructura sockadd_in. Aquí se almacenará información de la conexión entrante. Se utiliza para determinar que host está llamando y desde qué número de puerto.
- *(3º argumento, addrlen),* debe ser establecido al tamaño de la estuctura sockaddr. sizeof(struct sockaddr).

Función connect()

Inicia la conexión con el servidor remoto, lo utiliza el cliente para conectarse.

```
/* Iniciar conexión con un servidor */
int connect ( int sockfd, struct sockaddr *serv_addr, socklen_t addrlen )
```

Esta función admite tres parámetros:

- *(1º argumento, sockfd),* es el descriptor del socket devuelto por la función socket().
- *(2º argumento, serv_addr),* estructura sockaddr que contiene la dirección IP y número de puerto destino.
- *(3º argumento, serv_addrlen),* debe ser inicializado al tamaño de struct sockaddr. sizeof(struct sockaddr).

Funciones de Envío/Recepción

Después de establecer la conexión, se puede comenzar con la transferencia de datos. Podremos usar 4 funciones para realizar transferencia de datos.

```
/* Función de envío: send() */
send ( int sockfd,  void *msg, int len, int flags )
```

Esta función admite cuatro parámetros:

- *(1º argumento, sockfd),* es el descriptor socket por donde se enviarán los datos.
- *(2º argumento,* msg*),* es el puntero a los datos a ser enviados.
- *(3º argumento,* len*),* es la longitud de los datos en bytes.
- *(4º argumento, flags),* para ver las diferentes opciones consultar man send (la usaremos con el valor 0).

La función *send()* retorna la cantidad de datos enviados, la cual podrá ser menor que la cantidad de datos que se escribieron en el buffer para enviar.

```
/* Función de recepción: recv() */
recv ( int sockfd, void *buf, int len,  int flags )
```

Esta función admite cuatro parámetros:

- *(1º argumento, sockfd),* es el descriptor socket por donde se enviarán los datos.
- *(2º argumento, buf),* es el puntero a un buffer donde se almacenarán los datos recibidos.
- *(3º argumento, len),* es la longitud del buffer buf.

- *(4º argumento, flags),* para ver las diferentes opciones consultar man recv (la usaremos con el valor 0).

Si no hay datos a recibir en el socket , la llamada a recv() no retorna (bloquea) hasta que llegan datos. Recv() retorna el número de bytes recibidos.

```
/* Funciones de envío: write() */

write ( int sockfd, const void *msg, int len )
```

Esta función admite tres parámetros:

- *(1º argumento, sockfd),* es el descriptor socket por donde se enviarán los datos.
- *(2º argumento, msg),* es el puntero a los datos a ser enviados.
- *(3º argumento, len),* es la longitud de los datos en bytes.

```
/* Función de recepción: read() */

 read ( int sockfd, void *msg, in len )
```

Esta función admite los mismos parámetros que write, con la excepción de que el **buffer** de datos es donde se almacenará la información que nos envien.

Función close()

Finaliza la conexión del descriptor del socket. La función para cerrar el socket es close().

```
close (descriptor);
```

El argumento es el descriptor del socket que se desea liberar**.**

1.2 EJEMPLO DE SERVIDOR ITERATIVO

El siguiente programa es un ejemplo muy sencillo de un servidor orientado a conexión, es decir usando TCP, que proporciona un servicio de envío de caracteres orientado hasta que se recibe la cadena de caracteres “FIN”. El fichero está disponible en el moodle.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>


/*
 * El servidor ofrece el servicio de incrementar un número recibido de un
cliente
 */


main ( )
{

	/*-----------------------------------------------------
		Descriptor del socket y buffer de datos
	-----------------------------------------------------*/
	int sd, new_sd;
	struct sockaddr_in sockname, from;
	char buffer[100];
	socklen_t from_len;


	/* -----------------------------------------------------
		Se abre el socket
	-----------------------------------------------------*/
	sd = socket (AF_INET, SOCK_STREAM, 0);
	if (sd == -1)
	{
		perror("No se puede abrir el socket cliente\n");
		exit (1);
	}

	/*
	 * El servidor ofrece el servicio de incrementar un número recibido de
	   un cliente
	*/

	sockname.sin_family = AF_INET;
	sockname.sin_port = htons(2000);
	sockname.sin_addr.s_addr =  INADDR_ANY;

	if (bind (sd, (struct sockaddr *)&sockname, sizeof (sockname)) == -1)
	{
		perror("Error en la operación bind");
		exit(1);
	}


	/*-----------------------------------------------------------------
		Del las peticiones que vamos a aceptar sólo necesitamos el
		tamaño de su estructura, el resto de información (familia,puerto,
		ip), nos la proporcionará el método que recibe las peticiones.
	-----------------------------------------------------------------*/
	from_len = sizeof (from);
```

```
	if(listen(sd,1) == -1){
		perror("Error en la operación de listen");
		exit(1);
	}

	/*-----------------------------------------------------------------
		El servidor acepta una petición
	-----------------------------------------------------------------*/

	if((new_sd = accept(sd, (struct sockaddr *) &from, &from_len)) == -1){
		perror("Error aceptando peticiones");
		exit(1);
	}
	do
	{
		if(recv(new_sd, buffer, 100, 0) == -1)
			perror("Error en la operación de recv");

		printf("el mensaje recibido fue: \n%s\n", buffer );

	}while(strcmp(buffer, "FIN")!=0);


	close(new_sd);
	close(sd);

}
```

1.3 EJEMPLO DE CLIENTE

El siguiente programa es un ejemplo de cliente muy sencillo, que solicita un servicio orientado a conexión, de envío de caracteres orientado a conexión. El fichero está disponible en el moodle.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>


main ( )
{

	/*-----------------------------------------------------
		Descriptor del socket y buffer de datos
	-----------------------------------------------------*/
	int sd;
	struct sockaddr_in sockname;
	char buffer[100];
	socklen_t len_sockname;


	/* -----------------------------------------------
		Se abre el socket
	-----------------------------------------------*/
	sd = socket (AF_INET, SOCK_STREAM, 0);
	if (sd == -1)
	{
		perror("No se puede abrir el socket cliente\n");
		exit (1);
	}
```

```
        /* ------------------------------------------------------------------
              Se rellenan los campos de la estructura con la IP del
              servidor y el puerto del servicio que solicitamos
        ------------------------------------------------------------------*/
        sockname.sin_family = AF_INET;
        sockname.sin_port = htons(2000);
        sockname.sin_addr.s_addr =  inet_addr("127.0.0.1");

        /* ------------------------------------------------------------------
              Se solicita la conexión con el servidor
        ------------------------------------------------------------------*/
        len_sockname = sizeof(sockname);

        if (connect(sd, (struct sockaddr *)&sockname, len_sockname) == -1)
        {
              perror ("Error de conexión");
              exit(1);
        }

        /* ------------------------------------------------------------------
              Se transmite la información
        ------------------------------------------------------------------*/
        do
        {
                    puts("Teclee el mensaje a transmitir");
                    gets(buffer);

                    if(send(sd,buffer,100,0) == -1)
                          perror("Error enviando datos");

        }while(strcmp(buffer, "FIN") != 0);


        close(sd);
}
```

1.5 TERMINOLOGÍA Y CONCEPTOS DEL PROCESADO CONCURRENTE

Normalmente a un programa servidor se pueden conectar **varios clientes** simultáneamente. Hay dos opciones posibles para realizar esta tarea:

- Crear un nuevo proceso por cada cliente que llegue, estableciendo el proceso principal para estar pendiente de aceptar nuevos clientes.
- Establecer un mecanismo que nos avise si algún cliente quiere conectarse o si algún cliente ya conectado quiere algo de nuestro servidor. De esta manera, nuestro programa servidor podría estar "dormido", a la espera de que sucediera alguno de estos eventos.

La primera opción, la de múltiples procesos/hilos, es adecuada cuando las peticiones de los clientes son muy numerosas y nuestro servidor no es lo bastante rápido para atenderlas consecutivamente. Si, por ejemplo, los clientes nos hacen en promedio una petición por segundo y tardamos cinco segundos en atender cada petición, es mejor opción la de un proceso por cliente. Así, por lo menos, sólo sentirá el retraso el cliente que más pida.

La segunda es buena opción cuando recibimos peticiones de los clientes que podemos atender más rápidamente de lo que nos llegan. Si los clientes nos hacen una petición por segundo y tardamos un milisegundo en atenderla, nos bastará con un único proceso pendiente de todos. Esta opción será la que se implemente en la práctica, usando para ello la función *select()*.

Función select

```
/* Función de recepción:select() */

int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

- Los parámetros son:
  - *n: v*alor incrementado en una unidad del descriptor más alto de cualquiera de los tres conjuntos.
  - *readfds:* conjunto de sockets que será comprobado para ver si existen caracteres para leer. Si el socket es de tipo *SOCK_STREAM* y no esta conectado, también se modificará este conjunto si llega una petición de conexión.
  - *writefds:* conjunto de sockets que será comprobado para ver si se puede escribir en ellos.
  - *exceptfds:* conjunto de sockets que será comprobado para ver si ocurren excepciones.
  - *timeout: l*imite superior de tiempo antes de que la llamada a *select* termine. Si *timeout* es *NULL*, la función *select* no termina hasta que se produzca algún cambio en uno de los conjuntos (llamada bloqueante a *select*).

Para manejar el conjunto *fd_set* se proporcionan cuatro macros:

//Inicializa el conjunto *fd_set* especificado por *set*.
*FD_ZERO(fd_set *set);*

//Añaden o borran un descriptor de socket dado por *fd* al conjunto dado por *set*.
*FD_SET(int fd, fd_set *set);*
*FD_CLR(int fd, fd_set *set);*

//Mira si el descriptor de socket dado por *fd* se encuentra en el conjunto especificado por *set*.
*FD_ISSET(int fd, fd_set *est);*

Estructura timeval

```
/* Estructura timeval */

struct timeval
{
        unsigned long int tv_sec; /* Segundos */
        unsigned long int tv_usec; /* Millonesimas de segundo */
};
```

2. Enunciado de la práctica 2

Diseño e implementación de una versión simplificada de un servidor de IRC en modo consola. Internet Relay Chat (IRC) es un protocolo de aplicación que permite la conversación entre grupos de usuarios en Internet en tiempo real (síncrono) a través de Internet, teniendo las posibilidades de distintos grupos de usuarios, conversaciones privadas, existencia de moderadores en los grupos, etc. El IRC se definió originalmente en el RFC 1459, pudiendo encontrarse las actuales especificaciones en los RFC 2810, 2811, 2812 y 2813.

2.1 Descripción

La comunicación entre los clientes del chat se realizará bajo el protocolo de transporte TCP. La práctica que se propone consiste en la realización de una aplicación cliente/servidor que implemente un **servicio de chat** simplificado mediante el cual, los usuarios del chat (los clientes) se conectan al servicio de chat (el servidor) y pueden mandar y recibir mensajes de forma instantánea entre los distintos usuarios.

El procedimiento que se seguirá será el siguiente

- Un cliente se conecta al servicio y si la conexión ha sido correcta el sistema devuelve *+0k. Usuario Conectado*.
- Un cliente podrá participar en el chat indicando un mensaje de **"**JOIN *usuario***"**, siendo *usuario* el identificador que desea usar en el chat. Recibido este mensaje en el servidor, éste se encargará de enviar al nuevo usuario los usuarios que ya están conectados en ese momento (de este modo el usuario que se conecte tendrá conocimiento de quienes forman parte de la comunicación) y al resto de usuarios les informará del nuevo usuario que se ha conectado.
- Cada vez que un cliente mande un mensaje al servidor, éste deberá reenviarlo a su vez a todos los demás clientes conectados, sin incluir al cliente del que recibió el mensaje.
- Un mensaje de un cliente en la que se envíe la petición "*QUIT*", hará que el servidor le dé de baja y ya no le mande más mensajes, además le comunicará al resto de usuarios que dicho cliente ha abandonado la conversación.

Algunas de las restricciones a tener en cuenta son:

- La comunicación será mediante consola.
- Solamente se permite una única conversación, todos los usuarios que se conecten al chat participan en la misma conversación.
- El cliente deberá aceptar como argumento una dirección IP que será la dirección del servidor al que se conectará.
- El protocolo deberá permitir mandar mensajes de tamaño arbitrario. Teniendo como tamaño máximo envío una cadena de longitud 250 caracteres.
- El servidor aceptará servicios en el puerto 2050.
- El servidor debe permitir la conexión de varios clientes simultáneamente. Se utilizará la función *select()* para permitir esta posibilidad.
- El número máximo de clientes conectados en el chat será de 50 usuarios.

La segunda parte de la práctica consistirá en introducir la posibilidad de mantener varias salas de chats operativas, pudiendo decidir cada usuario registrado en qué sala desea participar. El procedimiento que se seguirá será el siguiente:

- Un cliente se conecta al servicio y si la conexión ha sido correcta el sistema devuelve "*+0k. Usuario conectado"*.
- Para poder acceder a los servicios es necesario identificarse mediante el envío del usuario y clave para que el sistema lo valide[1]. Los mensajes que deben indicarse son: "*USER usuario*" para indicar el usuario, tras el cual el servidor enviará "+*Ok. Usuario correcto*" o "*–ERR. Usuario incorrecto*". En caso de ser correcto el siguiente mensaje que se espera recibir es "*PASS password*", donde el servidor responderá con el mensaje de "+*Ok. Usuario validado*" o "*–ERR. Error en la validación*".
- Un usuario nuevo podrá registrarse mediante el mensaje "REGISTER –u usuario –p password –d Nombre y Apellidos –c Ciudad -t temas de interés". Se llevará un control para evitar colisiones con los nombre de usuarios.
- Una vez que el proceso de validación haya sido correcto, el servidor lo registra como cliente del chat y le envía los canales disponibles para comunicarse. El usuario indicará el canal de chat en la que desea participar mediante un mensaje "JOIN *nombreCanal*". Si el usuario desea abrir un nuevo canal podrá realizarlo

[1] El control de usuarios y claves en el servidor se llevará mediante un fichero de texto plano y no se codificará ningún tipo de encriptación.

mediante el mensaje “ADD *nombreCanal* –d *breveDescripcion*” a partir de este momento el servidor también ofrecerá este nuevo canal a los nuevos usuarios que se conecten y también se lo comunicará a los usuarios conectados en otros canales mediante un mensaje. El servidor también controlará que no haya más de un canal con el mismo identificador.

- Para abandonar un canal se usará el mensaje “*LEAVE canal*”, de este modo el usuario ya no puede mandar mensajes ni recibirlos con respecto a ese canal, pero podrá conectarse a otros canales. Si el usuario que abandona un canal era el último usuario que tenía el canal, este canal se eliminará de los canales disponibles.
- Para salir del servicio se usará el mensaje “*QUIT*”, de este modo el servidor lo eliminará de clientes conectados.
- Cualquier mensaje que no use uno de los especificadores detallados, se considerará un mensaje para compartir con los usuarios del canal.

Algunas restricciones a tener en cuenta en esta nueva versión que se debe implementar es:

- La comunicación será mediante consola.
- El cliente deberá aceptar como argumento una dirección IP que será la dirección del servidor al que se conectará.
- El protocolo deberá permitir mandar mensajes de tamaño arbitrario. Teniendo como tamaño máximo envío una cadena de longitud 250 caracteres.
- El servidor aceptará servicios en el puerto 2050.
- El servidor debe permitir la conexión de varios clientes simultáneamente. Se utilizará la función *select()* para permitir esta posibilidad.
- El número máximo de clientes conectados en el chat será de 50 usuarios.
- Siempre existirá un canal por defecto, el cual estará en funcionamiento mientras esté el servidor activo tenga o no clientes. Un máximo de 15 canales podrán estar activos a la vez, en caso de que un cliente desee crear un nuevo canal habiendo 15 canales activos no se permitirá esta acción y tendrá que esperar.
- Todos los mensajes mandados al servidor con respecto a la conexión y validación o la creación de canales recibirán una respuesta indicando que ha sido correcto “+OK. Texto informativo” o que ha habido algún error “-ERR. Texto informativo”.

En esta ampliación del chat, el servidor también llevará un control de la IP, la hora de conexión y de desconexión de cada usuario. Esta información se almacenará en ficheros de texto plano, generando un fichero por día, cuyo nombre será Día-Mes-Año (para obtener esta información el servidor hará uso a su vez del servidor de daytime de la máquina góngora.uco.es). No es necesario establecer una exclusión en la comunicación del fichero ya que la concurrencia es simulada y por tanto no van a acceder dos procesos a la vez al fichero.

Considerando la práctica completa, vamos a considerar los siguientes tipos de mensajes con el siguiente formato cada uno:

- **JOIN** *user*: mensaje para solicitar permiso para introducirse al canal de comunicación del chat (se considera en la primera versión donde solamente se considera un único canal).
- **USER** *usuario***:** mensaje para introducir el usuario que desea conectarse.
- **PASS** *contraseña***:** mensaje para introducir la contraseña asociada al usuario.
- **REGISTER** *–u usuario –p password –d Nombre y Apellidos –c Ciudad -t temas de interés*: mensaje mediante el cual el usuario solicita registrarse para acceder al servicio de chat que escucha en el puerto TCP 2050.
- **JOIN** *nombreCanal*: mensaje para solicitar permiso para introducirse a un canal concreto.
- **ADD** *nombreCanal –d breveDescripcion*: mensaje para añadir un nuevo canal al chat.
- **LEAVE** *canal***:** mensaje para solicitar salir de un canal.
- **QUIT:** mensaje para solicitar salir del chat.
- Cualquier otra línea que se escriba será reconocida por el protocolo como un mensaje a compartir con el resto de usuarios del canal de comunicación en el chat.

2.2 Objetivo

- Conocer las funciones básicas para trabajar con sockets y el procedimiento a seguir para conectar dos procesos mediante un socket.
- Comprender el funcionamiento de un servicio orientado a conexión y confiable del envío de paquetes entre una aplicación cliente y otra servidora utilizando sockets.
- Comprender las características o aspectos claves de un protocolo:
    - **Sintaxis:** formato de los paquetes
    - **Semántica:** definiciones de cada uno de los tipos de paquetes