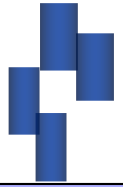
Práctica 1
Programación con Socket
Realización de cliente/servidor daytime

Amelia Zafra Gómez
Dpto. Informática y Análisis Numérico

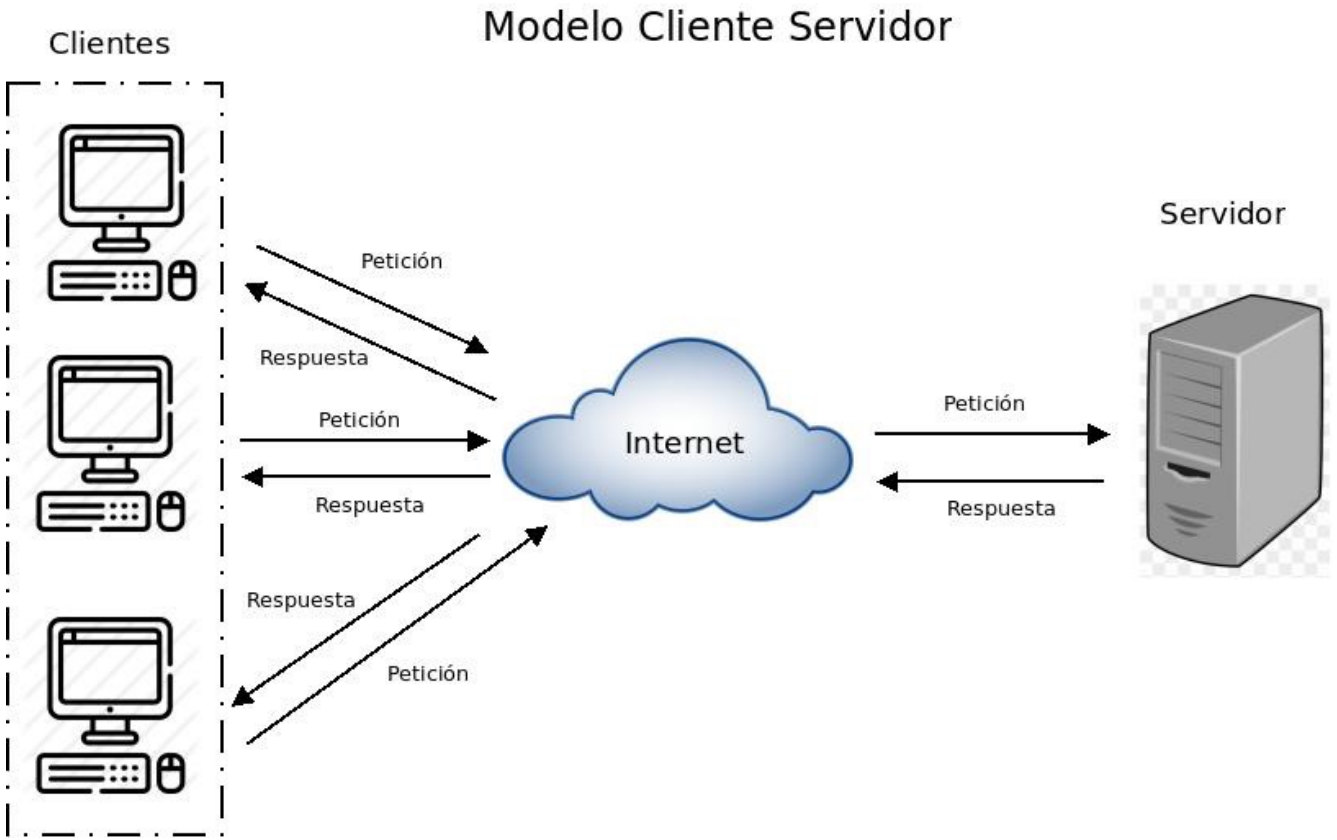


Práctica 1: Programación Socket

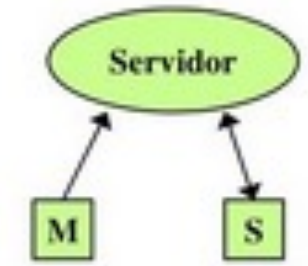
Fundamentos de la práctica 1

1. Estructura y funciones útiles en la interfaz socket
2 Ejemplo de aplicación
    2.1 Descripción de los pasos en el servidor
        2.1.1. Abrir el socket
        2.1.2 Asociar el socket con un puerto
            2.1.2.1 Rellenar la estructura sockaddr_in
        2.1.4 Envío y recepción de mensajes
            2.1.4.1 Recibir mensajes en el servidor
            2.1.4.2 Enviar un mensaje al cliente
    2.2 Descripción de los pasos en el cliente
        2.2.1 abrir el socket
        2.2.2 envío y recepción de mensajes
            2.2.2.1 enviar un mensaje al servidor
            2.2.2.2 recibir un mensaje del servidor

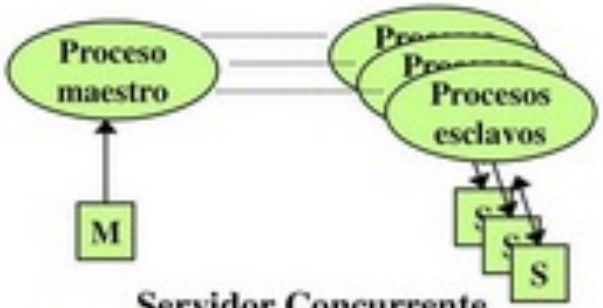
- Introducción
  - La programación de aplicaciones sobre TCP/IP se basa en el llamado modelo cliente-servidor.



- Introducción
  - Los servidores pueden clasificarse
    - Servidores concurrentes: atendiendo a si están diseñados para admitir múltiples conexiones simultáneas
    - Servidores iterativos: servidores que admiten una sola conexión simultánea.

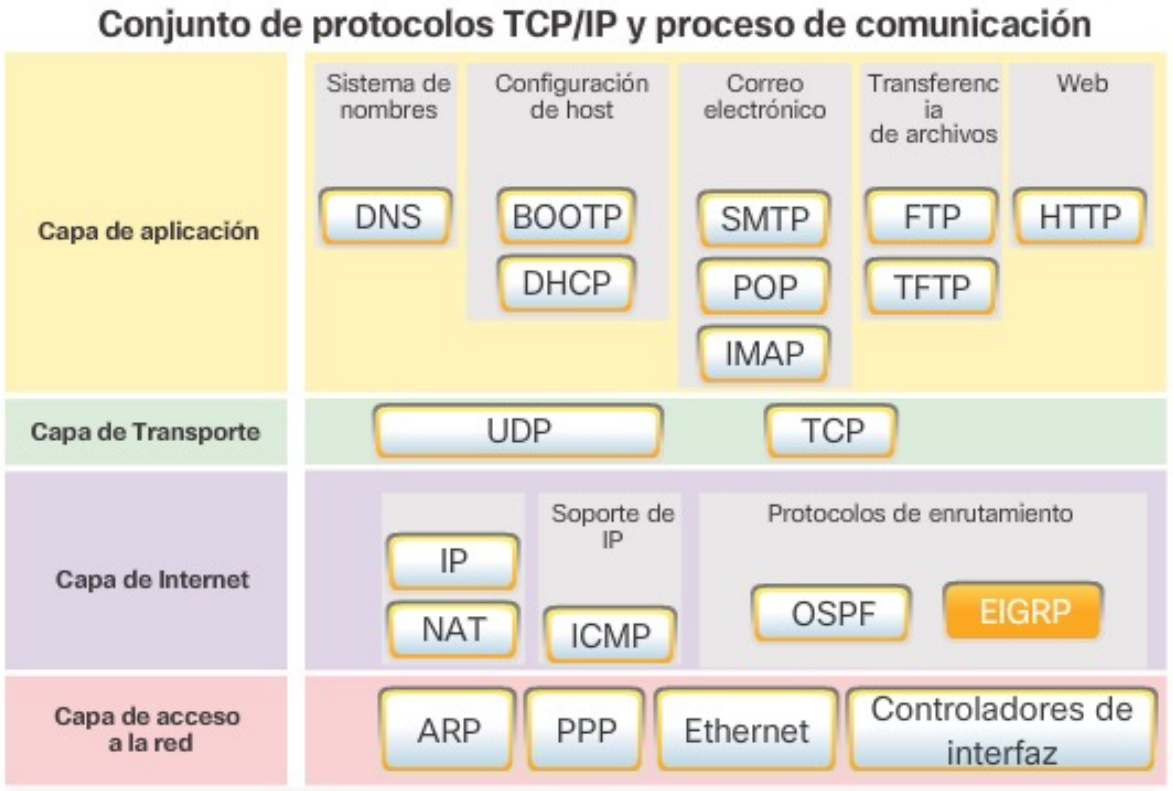


Servidor Iterativo

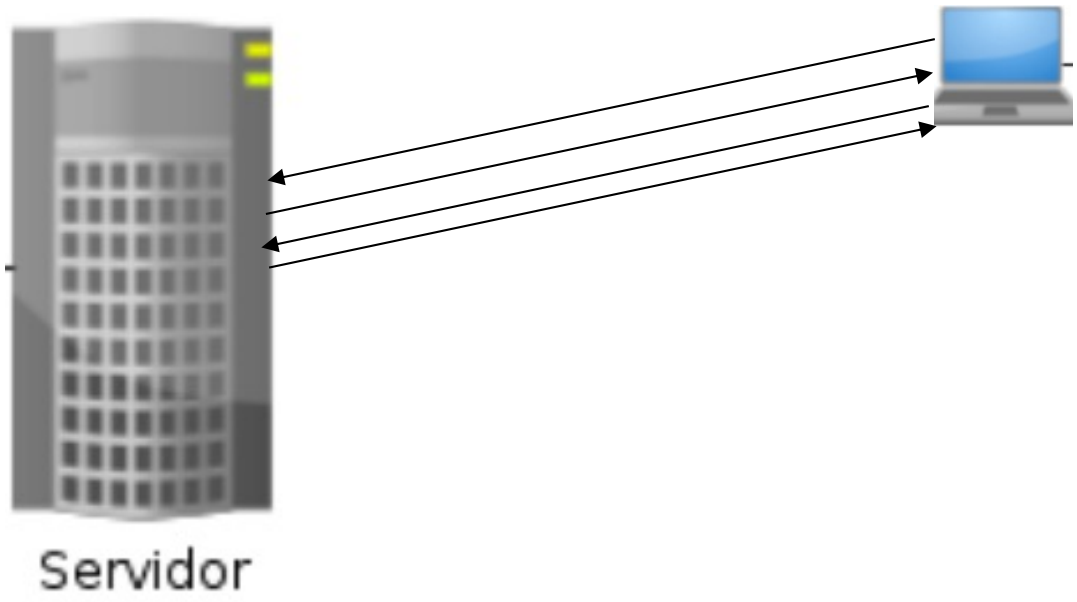


Servidor Concurrente

- Introducción
  - En función de la interacción entre el cliente y el servidor, se clasifican en orientada a conexión o no orientada a conexión, tenemos dos protocolos característicos de la capa de transporte: TCP y UDP.



- Introducción
  - Un servicio que sigue el modelo cliente-servidor se desarrolla en los siguientes pasos:



- Estructura y Funciones en la Interfaz Socket
  - Hay que incluir los siguientes ficheros de cabecera:

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

- Descripción de los pasos en el Servidor
  - *Los pasos que debe seguir un programa servidor son los siguientes:*
    - *Abrir un socket con la función socket()*
    - *Asociar el socket a un puerto con bind()*
    - *Leer mensaje con recvfrom()*
    - *Responder mensaje con sendto()*

- Abrir un socket con la función socket()
  - *Se abre el socket con la función socket(). Esta función nos devuelve un descriptor de socket. La forma de llamarla sería la siguiente:*

```
int Descriptor;
Descriptor = socket (AF_INET, SOCK_DGRAM, 0);
```

  - *El primer parámetro indica la familia, en nuestro caso será AF_INET.*
  - *El segundo indica que es UDP (SOCK_STREAM indicaría un socket TCP orientado a conexión).*
  - *El tercero es el protocolo que queremos utilizar. Hay varios disponibles, pero poniendo un 0 dejamos al sistema que elija este detalle.*

- Asociar el socket con un puerto
  - *En unix para el establecimiento de conexiones con sockets hay 65536 puertos disponibles, del 0 al 65535. Del 0 al 1023 están reservados para el sistema. El resto están disponibles.*
  - *Para decir al sistema operativo que deseamos atender a un determinado servicio, de forma que cuando llegue un mensaje por ese puerto nos avise, debemos llamar a la función bind().*

- Asociar el socket con un puerto

```
struct sockaddr_in Direccion;

/* Hay que rellenar la estructura Direccion */
Direccion = ... (la veremos a continuación)
bind ( Descriptor, (struct sockaddr *)&Direccion, sizeof (Direccion));
```

- *El primer parámetro es el descriptor de socket obtenido con la función socket().*
- *El segundo parámetro es un puntero a una estructura sockaddr que debemos rellenar adecuadamente. La Direccion la hemos declarado como struct sockaddr_in es una estructura más adecuada y es compatible con la estructura sockaddr (podemos hacer cast de una a otra). Para los sockets que comunican procesos en la misma máquina AF_UNIX, tenemos la estructura sockaddr_un, que también es compatible con sockaddr).*
- *El tercer parámetro es el tamaño de la estructura sockaddr_in.*

- Rellenar la estructura sockaddr_in

```
Direccion.sin_family = AF_INET;
Direccion.sin_port =htons(numero_puerto);
Direccion.sin_addr.s_addr = INADDR_ANY; (Direccion));
```

- El campo sin_family se rellena con el tipo de socket que estamos tratando, AF_INET en nuestro caso.

- El campo s_addr es la dirección IP que queremos asociar.

- Envío y recepción de mensajes
  - Con los pasos anteriores, el socket del servidor está dispuesto para recibir y enviar mensajes.
  - En el modelo cliente/servidor, el servidor lo primero que hace es ponerse a la escucha (recibir peticiones) y luego enviar mensajes con la respuesta.

- Recibir mensajes en el servidor
  - La función para leer un mensaje por un socket udp es *recvfrom()*.
  - La función se quedará bloqueada hasta que llegue un mensaje. Esta función nos devolverá el número de bytes leídos o -1 si ha habido algún error.

```
/* Contendrá los datos del que nos envía el mensaje */
struct sockaddr_in Cliente;

/* Tamaño de la estructura anterior */
int longitudCliente = sizeof(Cliente);

/* El mensaje es simplemente un entero, 4 bytes. */
int buffer;

recvfrom (Descriptor, (char *)&buffer, sizeof(buffer), 0, (struct sockaddr
*)&Cliente, &longitudCliente);
```

- Recibir mensajes en el servidor
  - *(1º argumento, int), es el descriptor del socket que queremos leer. Lo obtuvimos con socket().*
  - *(2º argumento, char *), es el buffer donde queremos que nos devuelva el mensaje. Podemos pasar cualquier estructura o array que tenga el tamaño suficiente en bytes para contener el mensaje. Debemos pasar un puntero y hacer el cast a char *.*
  - *(3º argumento, int),* es el número de bytes que queremos leer y que compondrán el mensaje. El buffer pasado en el campo anterior debe tener al menos tantos bytes como indiquemos aquí.
  - *(4º argumento, int),* son opciones de recepción. De momento nos vale un *0*.

- Recibir mensajes en el servidor
  - *(5º argumento, struct sockaddr),* esta estructura se pasa vacía y *recvfrom()* nos devolverá en ella los datos del que nos ha enviado el mensaje. Si los guardamos, luego podremos responderle con otro mensaje. Si no queremos responder, en este parámetro podemos pasar *NULL* (sabiendo que de este modo, no tenemos forma de saber quién nos ha enviado el mensaje ni de responderle).

  - *(6º argumento, int *),* debemos especificar con este parámetro el tamaño de la estructura *sockaddr_in*. La función nos lo devolverá con el tamaño de los datos contenidos en dicha estructura.

- Enviar mensajes al cliente
  - La función para envío de mensajes es *sendto()*.
  - La llamada envía el mensaje y devuelve el número de bytes escritos o -1 en caso de error.

```
/* Rellenamos el mensaje que se va a mandar con los datos que queramos
*/
buffer = ...;

sendto (Descriptor, (char *)&buffer, sizeof(buffer), 0, (struct sockaddr *)
&Cliente, longitudCliente);
```

- Enviar mensajes al cliente
  - *(1º argumento, int),* con el descriptor del socket por el que queremos enviar el mensaje. Lo obtuvimos con socket().
  - *(2º argumento, char *),* con el buffer de datos que queremos enviar. En este caso, al llamar a sendto() ya debe estar relleno con los datos a enviar.
  - *(3º argumento, int),* con el tamaño del mensaje anterior, en bytes.
  - *(4º argumento, int),* existen diferentes opciones. De momento nos vale poner un 0.
  - *(5º argumento, struct sockaddr),* en este caso, como es respuesta a la petición del cliente, podemos especificar la misma estructura que nos rellenó la función recvfrom(), estaremos enviando el mensaje al cliente que nos lo envío a nosotros previamente.
  - *(6º argumento, int),* con el tamaño de la estructura sockaddr. Vale el mismo entero que nos devolvió la función recvfrom() como sexto parámetro.

- Pasos en el Cliente
  - *Los pasos que debe seguir un programa cliente son los siguientes:*
    - ***Abrir un socket con*** *socket()*
    - ***Enviar mensaje al servidor con*** *sendto()*
    - ***Leer respuesta con*** *recvfrom()*
    - ***Cerrar el socket con*** *close()*
- Abrir un socket con la función socket()
  - Se trata de la misma especificación que en el cliente.

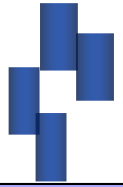
- Enviar mensajes al servidor
  - La función para envío de mensajes es *sendto()*.
  - La llamada envía el mensaje y devuelve el número de bytes escritos o -1 en caso de error.
  - Los parámetros y forma de funcionamiento es igual que en el servidor.
  - Para llamar a esta función tenemos que rellenar la estructura *sockaddr* del quinto parámetro, indicando los datos del servidor

```
Direccion.sin_family = AF_INET;
Direccion.sin_port = ...;   (El puerto del servicio)
Direccion.sin_addr.s_addr = ...; (La IP del servidor)
```

- Recibir mensajes en el cliente
  - La función para leer un mensaje por un socket udp es *recvfrom()*.
  - La función se quedará bloqueada hasta que llegue un mensaje. Esta función nos devolverá el número de bytes leídos o -1 si ha habido algún error.
  - El procedimiento es exactamente el mismo que el explicado en el servidor.

```
/* Contendrá los datos del que nos envía el mensaje */
struct sockaddr_in Servidor;
/* Tamaño de la estructura anterior */
int longitudServidor = sizeof(Servidor);
/* El mensaje es simplemente un entero, 4 bytes. */
int buffer;

recvfrom (Descriptor, (char *)&buffer, sizeof(buffer), 0, (struct sockaddr
*)&Servidor, &longtudServidor);
```



Práctica 1: Programación Socket

Enunciado de la práctica 1

1. Descripción
2. Objetivo
3. Representación
4. Funcionamiento

La práctica 1 consiste en el desarrollo de un protocolo de aplicación similar al servicio proporcionado por DAYTIME, que se ejecute por encima de UDP. Será necesario el diseño de un servidor/cliente UDP de DAYTIME

- Función select, comprueba el estado de un socket.

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

- Los parámetros son:
  - *n* Valor, incrementado en una unidad, del descriptor más alto de cualquiera de los tres conjuntos.
  - *readfds* Conjunto de sockets que serán comprobados 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 serán comprobados para ver si se puede escribir en ellos.
  - *exceptfds* Conjunto de sockets que serán comprobados para ver si ocurren excepciones.
  - *timeout* Limite 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).

- Función select, comprueba el estado de un socket.

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 timeout (select)

Campos:

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

Ejemplo de uso:

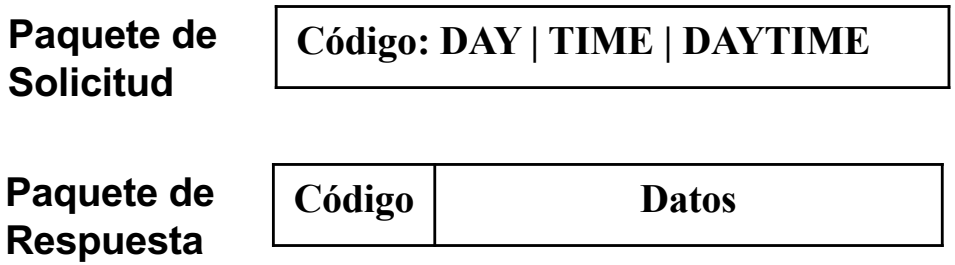
```
timeout.tv_sec=1;
timeout.tv_usec=0;
```

Uso UDP:

```
select ( n+1, &conjunto ,NULL,NULL, &timeout )
```

- Descripción
- Se pide diseñar un servidor UDP que implemente un servicio similar al proporcionado por daytime, así como un cliente que haga uso de su servicio.
- Como el servicio que ofrece UDP es no fiable tanto la petición como la respuesta pueden perderse, por lo que el cliente puede no recibir contestación.
  - El programa cliente realizará una petición al servidor y esperará la respuesta un tiempo limitado (5 segundos).
  - Si recibe la respuesta, enviará a la salida estándar el día y la hora proporcionados por el servidor.
  - Si después de ese tiempo no recibe una respuesta, volverá a mandar el mensaje al servidor para intentar recibir, si después de tres intentos no consigue nada, mostrará al usuario un mensaje de error.

- Descripción de los paquetes
- El servicio admitirá la siguiente especificación de los paquetes:
  - **DAY**: paquete para solicitar el día de la maquina remota. La salida será con el siguiente formato: *Martes, 21 de Septiembre de 2021.*
  - **TIME:** paquete para solicitar la hora de la maquina remota. La salida será con el siguiente formato: *18:00:00*
  - **DAYTIME:** paquete para solicitar tanto el día como la hora de la maquina remota. La salida será con el siguiente formato: *Martes,21 de Septiembre de 2021; 18:00:00.*



- Descripción de los paquetes
- El funcionamiento bajo UDP es el siguiente:
  - El cliente envía un datagrama UDP con el tipo de servicio que desea al servidor.
  - El servidor recibe el datagrama UDP, a partir del cual obtiene la dirección IP y el puerto UDP del cliente que solicita la información.
  - El servidor realiza una consulta para obtener la información solicitada por el cliente. Se puede hacer uso de las funciones *strftime* y *asctime* para obtener la información necesaria para ofrecer este servicio.
  - El servidor crea un datagrama UDP con la información obtenida en el punto 2, datagrama que tiene como único dato la cadena de texto a devolver.
  - El datagrama UDP llega al cliente, el cual puede leerlo y extraer la información del día y hora del servidor.

- Objetivos
  - Comprender el funcionamiento de un servicio iterativo no orientado a la conexión confiable y no confiable a través del envío de paquetes entre una aplicación cliente y otra servidor utilizando sockets.
  - Comprender como se añade confiabilidad al servicio proporcionado.
  - Comprender las características o aspectos claves de un protocolo:
    - **Sintaxis:** formato de los paquetes
    - **Semántica:** definiciones de cada uno de los tipos de paquetes