Redes y Comunicación (I310)

BSD Sockets

Universidad de SanAndrés

Práctica: Matsunaga - Grisales

Universidad de SanAndrés

Aprenderemos a responder estas preguntas

- ¿Qué es un socket?
- ¿Qué es una *system call*?
- ¿Qué tipos de socket existen? ¿Para qué se usa el port number?
- ¿Cuál es workflow para UDP y para TCP?
- ¿Cómo uso un sniffer en consola de linux?

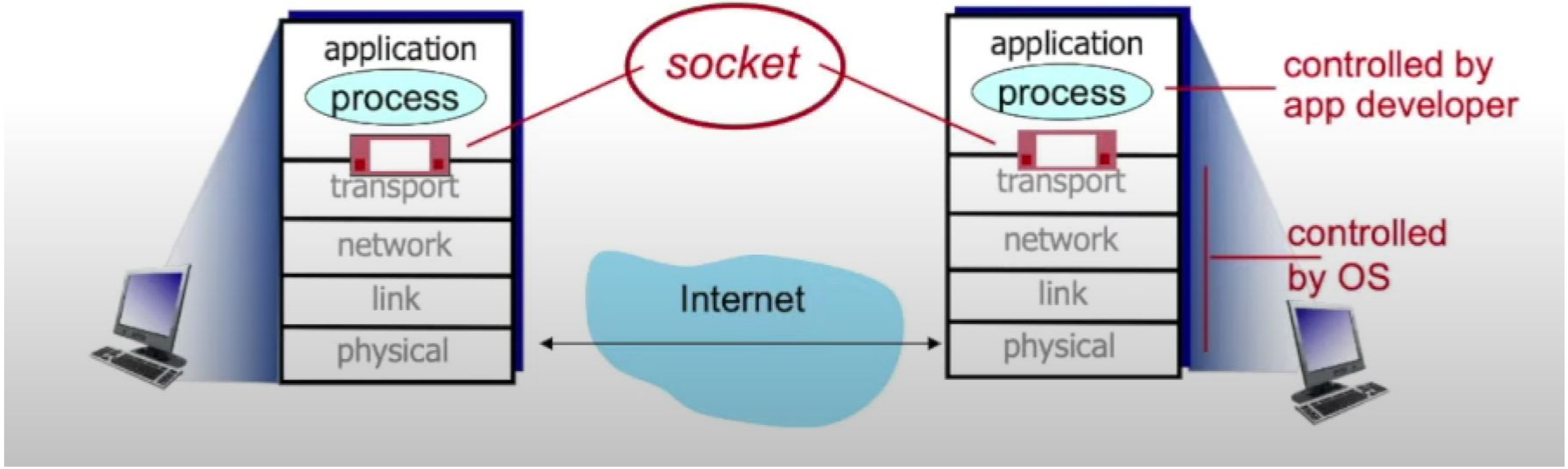
A leer la documentación https://beej.us/guide/bgnet/html/split-wide/ 😄

¿Qué es un socket?

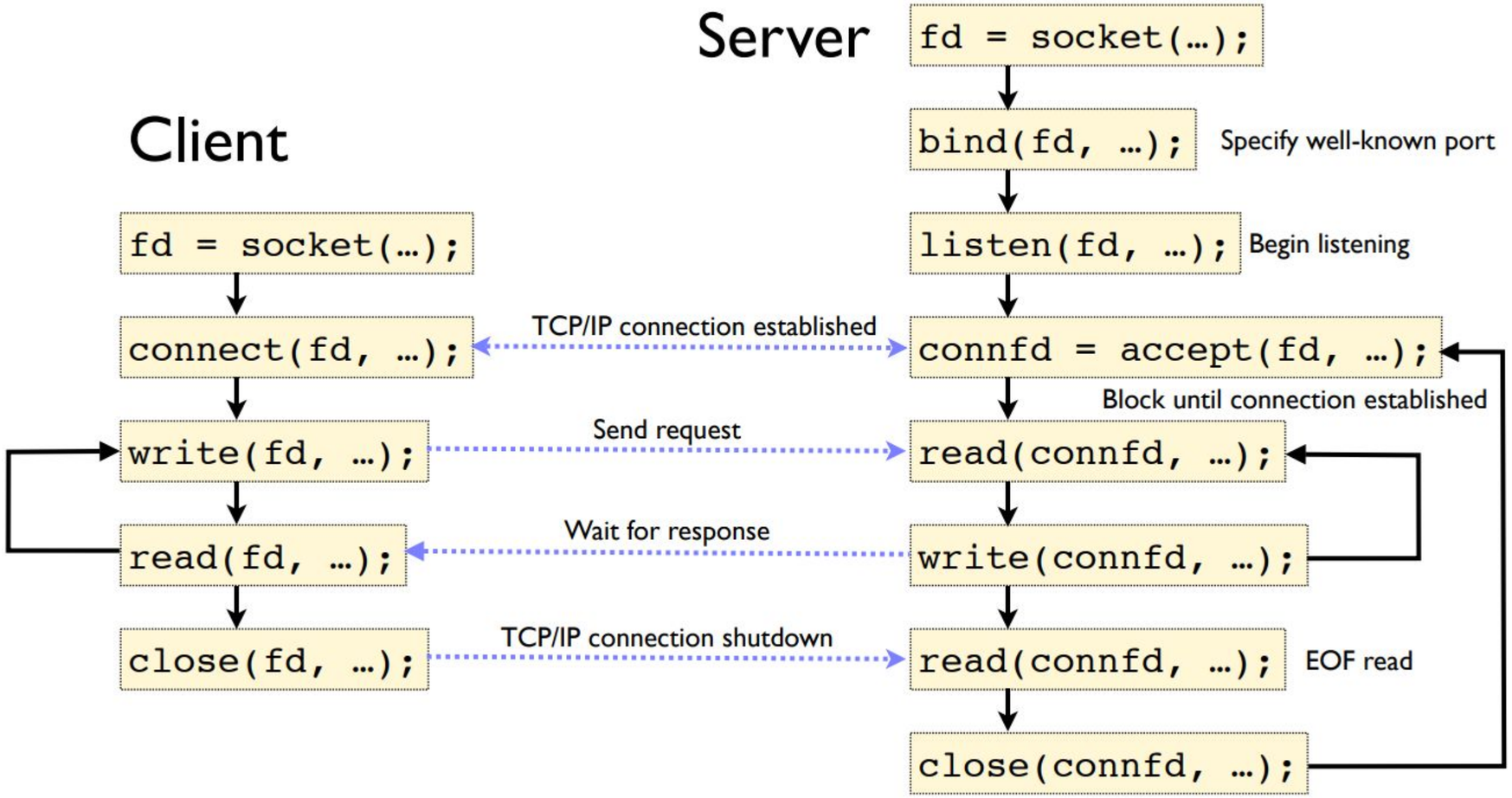
Es una manera de hablar con otros programas utilizando Unix *file descriptors*.

BSD Sockets (Berkeley Software Distribution) creados en 1983
Cross-platform: Linux, Mac, Windows (Winsockets), FreeBSD, Solaris, etc

BSD Sockets y la arquitectura en capas



TCP workflow



Un breve repaso de C: pasaje de argumentos

Universidad de SanAndrés

- En C el programa debe contener la función main()

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  /* argc da la cantidad de argumentos pasados */
  if (argc < 2 ) /* Si no se pasan dos argumentos sale */
  {
    printf("Debe ingresar al menos 1 parametro\n");
    fflush(stdout);
    return 1;
  }
  else
  {
    int i=0;
    for ( i=0; i<argc; i++)
    printf("El argumento nro %d es \"%s\"\n",i,argv[i]);
  }
  return 0;
}
```

- Para compilar: `gcc ex01-00-args.c -Wall -o ex01-00-args`

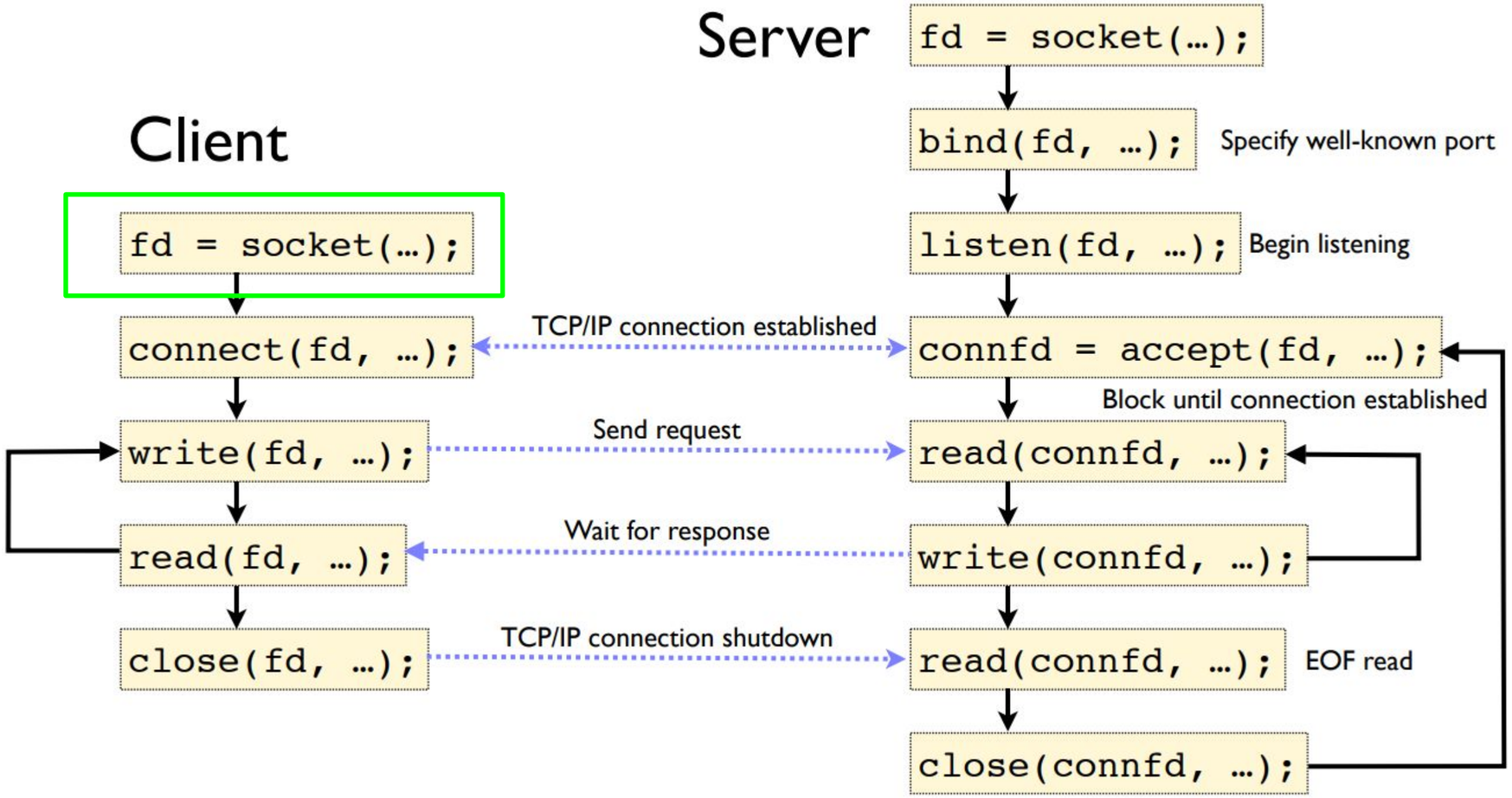
Un breve repaso de C: pasaje de argumentos (2)

Universidad de SanAndrés

- `argc` va de 1 en adelante (siempre cuenta el nombre del archivo ejecutable)
- `argv[0]` apunta a un C-string con la ruta de ejecución del programa
  - ¿usos?
  - ver ex01-01-args-name.c

    y crear un soft link al programa compilado: `ln -s ex01-01-args-name mi_hack`
- `argv[1]` apunta a un C-string con el primer argumento
- Los argumentos se delimitan por espacios en blanco

Creación de un socket



Creación de un socket

Universidad de SanAndrés

- El system call `socket` crea sockets cuando se lo pide. Toma tres argumentos enteros y devuelve un resultado entero.
  `resultado = socket(pf, tipo, protocolo)`
  - `pf` : especifica familia de protocolo que se va a utilizar, es decir, especifica cómo interpretar la dirección cuando se suministra (para TCP/IP es `PF_INET`)
  - `tipo`: especifica el tipo de comunicación que se desea. `SOCK_STREAM` es para servicio de entrega confiable de flujo (TCP), `SOCK_DGRAM` es para entrega de datagramas sin conexión (UDP)
  - `protocolo`: se utiliza para acomodar los diversos protocolos dentro de una familia. Especifica el protocolo en caso que el modelo de red soporte diferentes tipos de modelos de Stream y Datagrama. Como TCP/IP sólo tiene uno para cada uno, se setea en 0.

Creación de un socket (2)

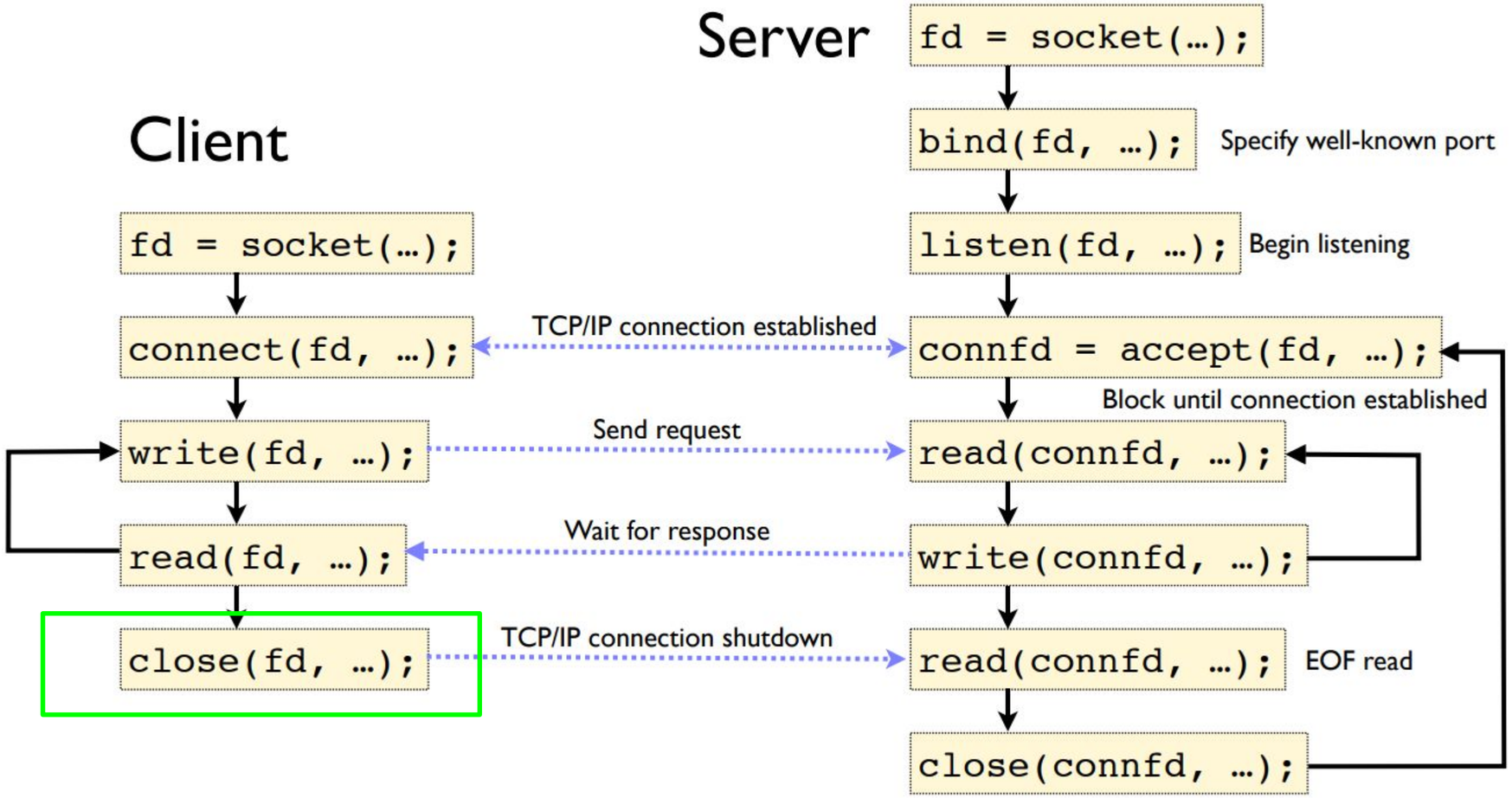
Universidad de SanAndrés

- Un ejemplo con TCP sería: `s = socket(PF_INET, SOCK_STREAM, 0)`
- Requiere los siguientes headers

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
int main(int argc, char *argv[])
{
  int s; /*file descriptor*/
  s = socket( PF_INET, SOCK_STREAM, 0);
  return 0;
}
```

- Ver ex02-00-socket-create.c

Finalización de un socket



Finalización de un socket

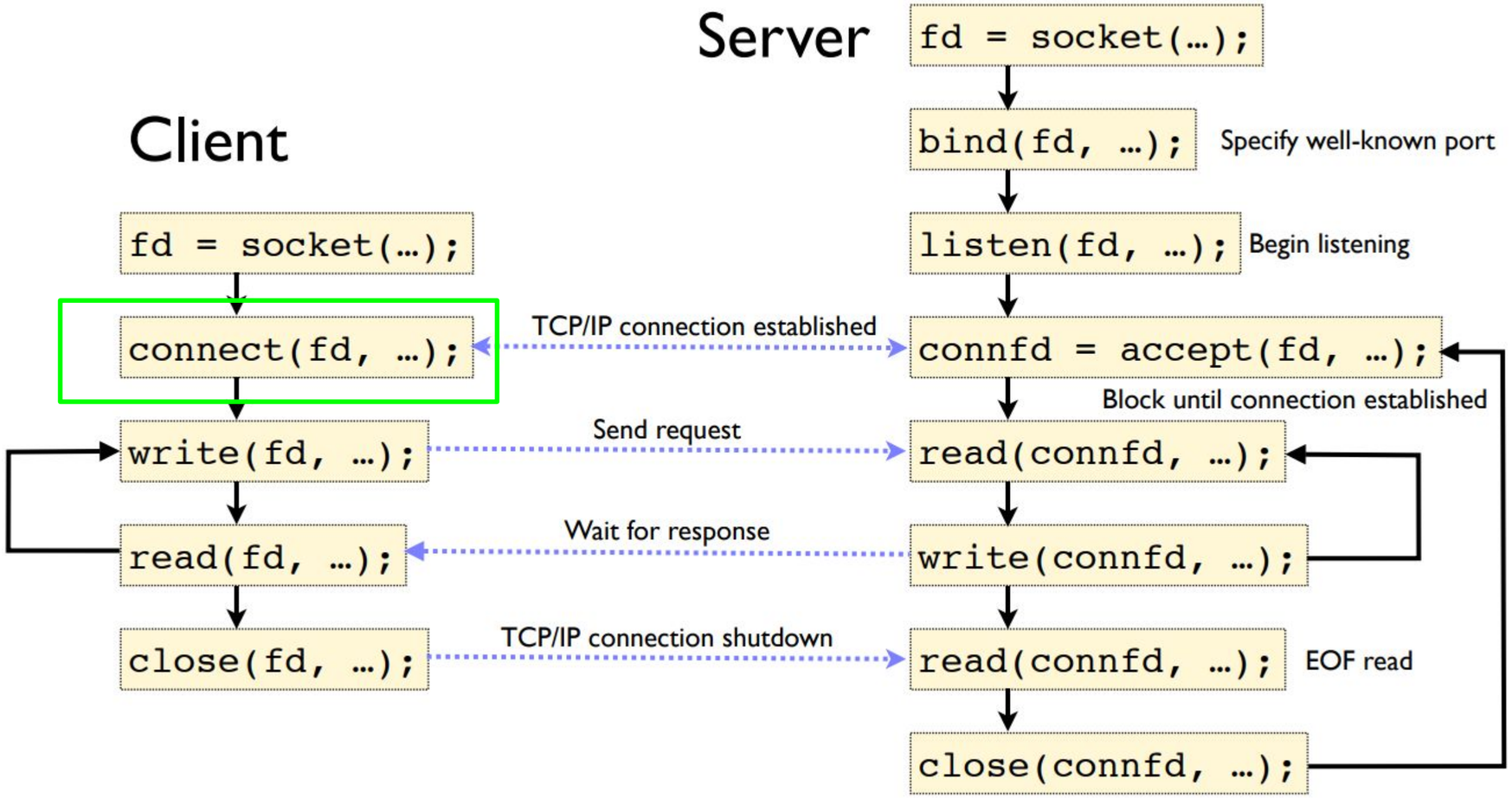
Universidad de SanAndrés

- Cuando un proceso termina de utilizar un socket, se llama a: `close(socket)`
- El argumento socket especifica al file descriptor (integer) de un socket para que cierre. (Ver ex02-01-socket-close.c)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
int main(int argc, char *argv[])
{
  int s; /*file descriptor*/
  s = socket( PF_INET, SOCK_STREAM,0);
  close(s);
  return 0;
}
```

- `shutdown()` permite controlar la finalización del socket para cada sentido independientemente

Conexión de un socket (TCP)



Conexión de un socket (TCP)

Universidad de SanAndrés

- un socket se crea en el ***unconnected state***
- El system call `connect` enlaza un destino permanente a un socket, colocándolo en el ***connected state***
- La aplicación debe llamar a connect para establecer una conexión antes de que pueda transferir datos si es `SOCK_STREAM`
- Los sockets `SOCK_DGRAM` no necesitan estar conectados antes de usarse, pero haciéndolo así, es posible transferir datos sin especificar el destino en cada ocasión
- `connect(socket, destaddr, addrlen)`
  - `socket`: file descriptor del socket
  - `destaddr`: es un `struct sockaddr` de socket en la que se especifica la dirección de destino a la que deberá enlazarse el socket.
  - `addrlen`: longitud de la dirección de destino medida en octetos.

Conexión de un socket (TCP) (2)

Universidad de SanAndrés

- En el caso de servicio sin conexión, `connect` no hace nada más que almacenar localmente la dirección de destino.

- The only purpose of `struct sockaddr` is to cast the structure pointer passed in `destaddr` in order to avoid compiler warnings.

- Ver ex03-00-connect-tcp.c

Conexión de un socket (TCP) (3)

Universidad de SanAndrés

```
#include <string.h> /* necesario para memset() */
#include <errno.h>  /* necesario para codigos de errores */
#include <netinet/in.h>
#include <netdb.h>  /* necesario para getaddrinfo() */
...
  struct addrinfo hints;
  struct addrinfo *servinfo; /* es donde van los resultados */
  int status;
  memset(&hints,0,sizeof(hints));  /* es necesario inicializar con ceros */
  hints.ai_family = AF_INET;       /* Address Family */
  hints.ai_socktype = SOCK_STREAM; /* Socket Type */
  if ( (status = getaddrinfo(argv[1],"80", &hints, &servinfo))!=0)
  {
fprintf(stderr, "Error en getaddrinfo: %s\n",gai_strerror(status));
return 1;
  }
  switch (connect(s, servinfo->ai_addr, servinfo->ai_addrlen))
  {
case -1:
   . . .
case 0:
   . . .
  }
  close(s);
}
```

Conexión de un socket (TCP) (4)

Universidad de SanAndrés

- `getaddrinfo()` nos simplifica la vida para no tener que lidiar con esas estructuras. A su vez hace resolución de nombres y servicios.

- Incluso nos ayuda a configurar un socket

```
s = socket( servinfo->ai_family, servinfo->ai_socktype, servinfo->ai_protocol);
```

- Ver ex03-01-connect-tcp-getaddrinfo.c

“Conexión”  de un socket (UDP)

Universidad de SanAndrés

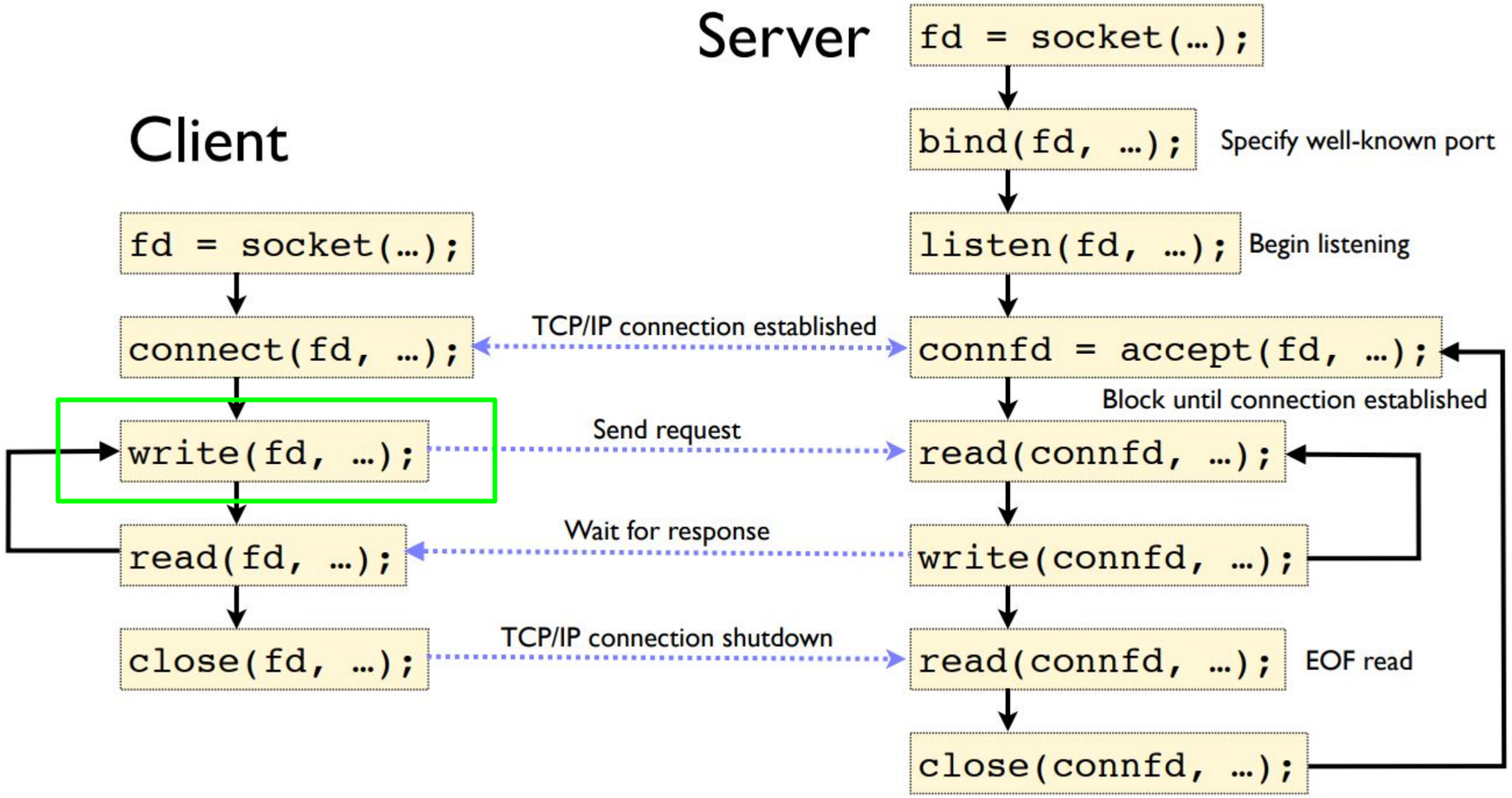
- Primero que nada UDP es NO orientado a la conexión. Llamamos así el tema sólo por la función `connect()`.

- Por lo tanto no realiza ningún intercambio de paquetes la ejecución de dicha función.

- Sí **fija el destino** (IP y puerto) para no tener que especificarlo repetidas veces.

- Ver ex03-02-connect-udp.c

Escribiendo en un socket



Escribiendo en un socket

Universidad de SanAndrés

- Hay 5 system calls que permiten escribir en socket

- “Típicas” de SOCK_STREAM
  - `write(socket, buffer, length)`
  - `send(socket, message, length, flags)`
  - ver ex04-00-send-tcp.c, ex04-01-write-tcp.c

- “Típicas” de SOCK_DGRAM
  - `sendto(socket, message, length, flags, destaddr, addrlen)`
  - `sendmsg(socket, messagestrtuct, flags)`
  - ver ex04-03-write-udp.c, ex04-04-write-sendto-udp.c
- La quinta es `writev` que no describiremos

Escribiendo en un socket (2)

Universidad de SanAndrés

- Por ejemplo, para mandar un request HTTP de una sola línea uso

```
write(s,argv[2],strlen(argv[2]));
write(s,"\r\n\r\n",4);
```

- o usando `send` pero sin usar flags

```
send(s,argv[2],strlen(argv[2]), 0);
send(s,"\r\n\r\n",4, 0);
```

- si usé `connect()` ambos ejemplos me sirven para UDP (por eso lo de “típicas”)

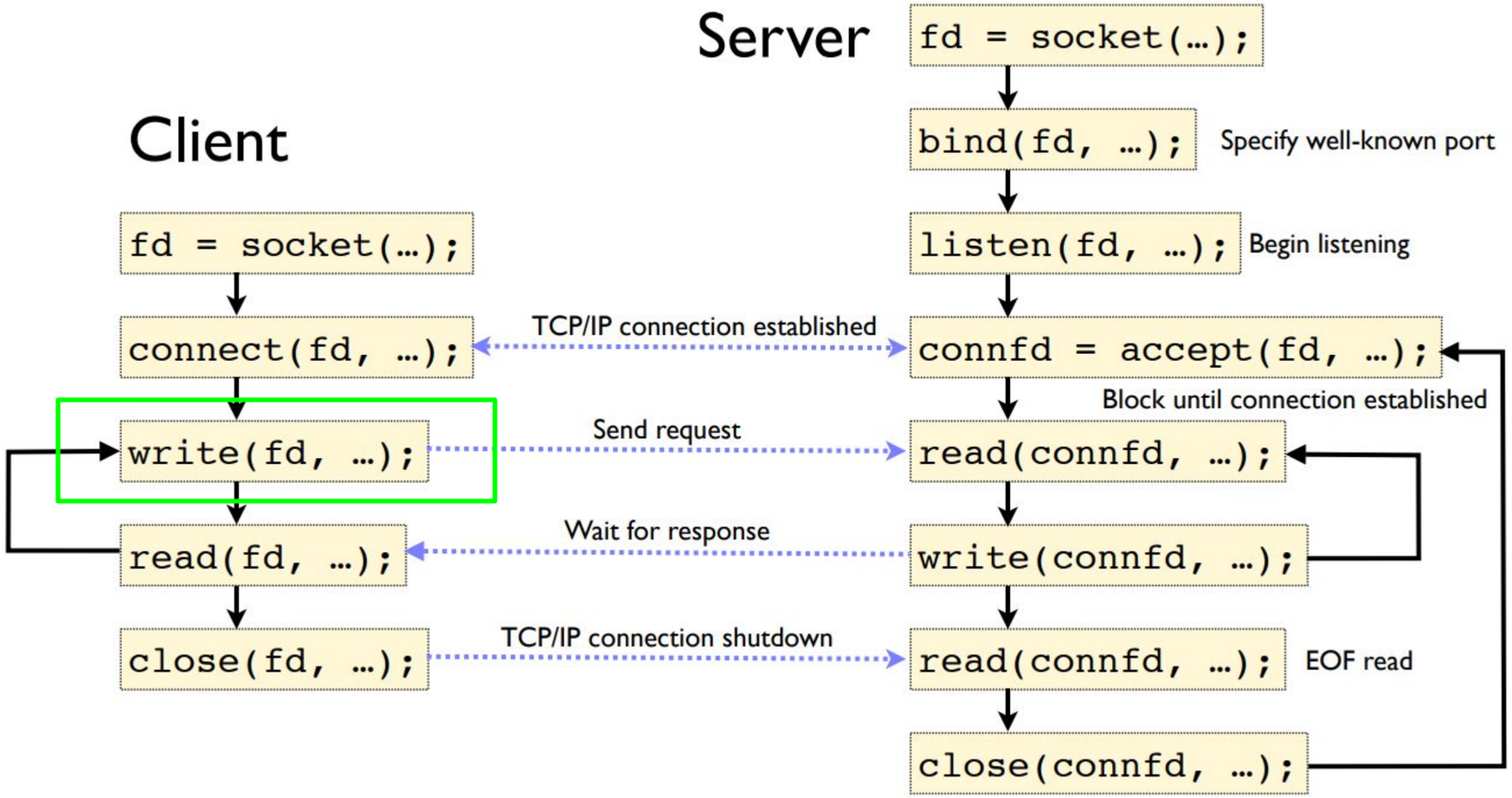
Escribiendo en un socket (3)

Universidad de SanAndrés

- Si uso `sendto()` habiendo hecho `connect()` sólo funcionará si la dirección especificada en el `struct sockaddr` es la misma que la que se usó en `connect()`

- `sendmsg()` queda como tarea (usar man sendmsg e Internet)

Lectura en un socket



Lectura en un socket

Universidad de SanAndrés

- También hay 5 system calls respectivamente

  `read(socket, buffer, length)  - write`

  `recv(socket, buffer, length, flags) - send`

  `recvfrom(socket, buffer, length, flags, fromaddr, addrlen) - sendto`

  `recvmsg(socket, messagestruct, flags) - sendmsg`

  `readv` que tampoco describiremos
- Al esperar datos siempre surge la dicotomía de usar **lecturas bloqueantes** o no. Nosotros no entraremos en ese detalle pero es importante saber que existe.
- A su vez es normal que aparezcan loops en la lectura de datos

Lectura en un socket (2)

Universidad de SanAndrés

- Ejemplo

```
int inbytes=-1;
char buffer[3000];
/* loop de lectura */
while (1)
{
  fprintf(stderr,"socket descriptor: %d inbytes=%d\n", s, inbytes);
  if (inbytes==0) /* cuando read devuelve 0 implica EOF */
  {
    close(s);
    return 1;
  }
  memset(buffer,0,sizeof(buffer));
  inbytes = recv(s, buffer, sizeof(buffer),0);
  printf("%s\n", buffer);
}
```

Lectura en un socket (3)

Universidad de SanAndrés

- Aquí leeremos hasta que de EOF (End Of File), en TCP eso ocurre cuando finalizan la conexión en el sentido que recibimos datos.

- En UDP es más complicado porque no hay señalización de cuando “terminan” los datos.

Cuándo leer y cuándo escribir

Universidad de SanAndrés

- Dijimos que la lectura es bloqueante, es decir que si intentamos leer y no hay datos el programa se **quedará trabado** hasta que haya datos.
- En el caso de la escritura surge un problema similar, ya que la escritura también es bloqueante por default, aunque este **problema es menos grave**
- Para evitar estas situaciones usando sockets bloqueantes existen funciones como `select()` y `poll()`. Estas permiten conocer si hay un file descriptor listo para escribir o leer dentro de un timeout especificado.
- Ver ex05-00-tcp-read-loop.c, ex05-01-udp-read-loop-fail.c, ex06-00-tcp-reset.c

Cuándo leer y cuándo escribir

Universidad de SanAndrés

- Ver ex06-00-tcp-reset.c
  - Generar un server en el port 81 con netcat

    ```
    # nc -l 81
    ```
  - Lanzar el client que le resetarán la conexión

    ```
    $ ./ex06-00-tcp-reset 127.0.0.1 81 "hola que tal"
    ```

Binding de un socket

Universidad de SanAndrés

- Hacer ***binding*** es ligar el socket a una dirección IP y puerto. Esto puede ser tanto para sockets que recibirán datos y estarán escuchando en dicha IP y puerto como para sockets que iniciarán una conexión (caso menos usado)

- Es el paso anterior y necesario al ***listening***

```
bind(socket, bindaddr, addrlen)
```

Binding de un socket (2)

Universidad de SanAndrés

```
memset(&hints,0,sizeof(hints));      /* es necesario inicializar con ceros */
hints.ai_family = AF_INET;           /* Address Family */
hints.ai_socktype = SOCK_STREAM;     /* Socket Type */
hints.ai_flags = AI_PASSIVE;         /* Llena la IP por mi por favor */
if ( (status = getaddrinfo(NULL, "1280", &hints, &servinfo))!=0)
{
  fprintf(stderr, "Error en getaddrinfo: %s\n",gai_strerror(status));
  return 1;
}
/* Creación del socket */
s = socket( servinfo->ai_family, servinfo->ai_socktype, servinfo->ai_protocol);
/* Binding del socket al puerto e IP configurados por getaddrinfo */
if (status = bind(s, servinfo->ai_addr, servinfo->ai_addrlen))
{
  fprintf(stderr, "Error en bind: %s\n",gai_strerror(status));
  return 1;
}
```

Listening de un socket

Universidad de SanAndrés

- Una vez hecho el binding solo hace falta poner a escuchar pedidos entrantes: `listen(socket, backlog)`
- Esta operación no bloquea si no hay peticiones, a diferencia de `accept()`
- Ejemplo

```
if (status = listen(s, 10)) /*Soporta hasta 10 pedidos de conexion en cola*/
{
  fprintf(stderr, "Error en listen: %s\n",gai_strerror(status));
  return 1;
}
```

- Ver ex07-00-tcp-listen.c

Aceptando conexiones

Universidad de SanAndrés

- Aceptar una conexión devuelve como resultado un **nuevo** socket

```
new_s = accept(socket, their_addr, addrsize)
```

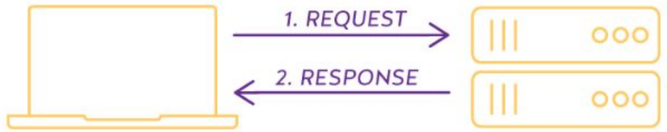
- Ejemplo

```
int new_s;
struct sockaddr their_addr;
socklen_t addrsize;
addrsize = sizeof their_addr;
if ((new_s = accept(s, &their_addr, &addrsize))==-1)
{
  fprintf(stderr, "Error en accept: %s\n",gai_strerror(status));
  return 1;
}
```

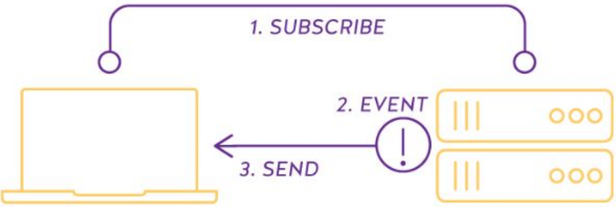
- La dirección remota se almacenará en la estructura `sockaddr` y su longitud será `socklen_t`, la cual sólo será alterada de ser necesario (sólo si tenemos IPv4/IPv6)
- Ahora con ese nuevo socket podemos leer y escribir. Ver ex07-00-tcp-listen.c

Paradigmas de interacción

- Client-server
  - ***server***: *always-on* host,
    - atiende peticiones (*requests*) de los clientes (*clients)*
  - ***client***: host espóradicamente conectado.
    - Cuando necesita información hace una petición
  - Ejemplo: navegar con un browser en Internet

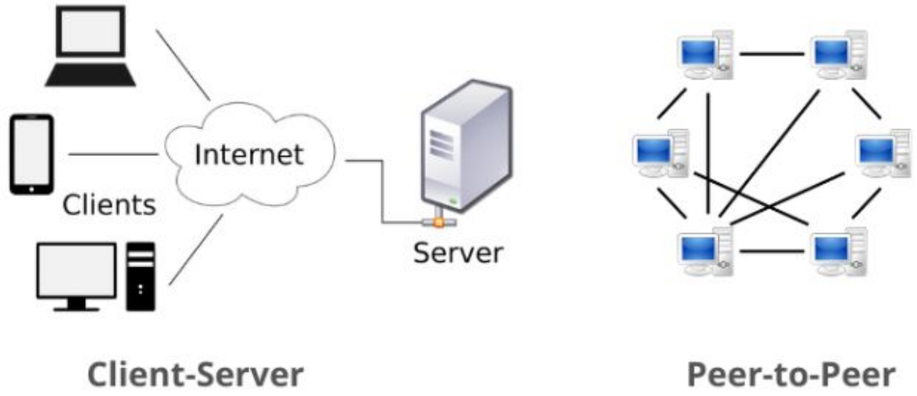


- Publisher-subscriber
  - ***subscriber***: se suscribe a una entidad (*event bus/message broker*)
  - ***publisher***: entidad que genera mensajes y los publica (a un *event bus/message broker*)
  - ***event bus/message broker***: se encargar de hacer llega la información a quién esté interesado (suscrito)
  - se delega la responabilidad: en lugar de esperar una petición del cliente, el cliente expresa la intención de recibir información a futuro sobre un tópico.



Paradigmas de interacción

- Peer-to-peer
  - ***peer***: es cliente/servidor a la vez, hace peticiones y atiende peticiones
  - Descentralizado, la tarea suele dividirse en partes, requiere de mecanismos de ***rendezvous***
  - Ejemplo: p2p file sharing systems (bittorrent, edonkey, etc), cryptocurrencies



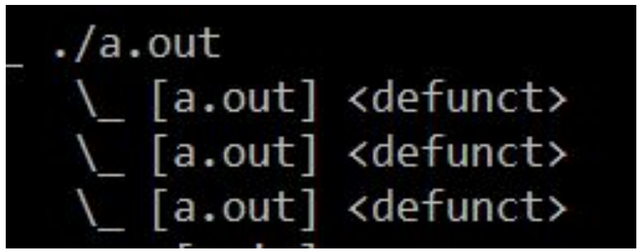
Arquitectura de server

- Notamos que si hacemos un programa lineal y simple “no podemos” atender varios clientes concurrentemente
- Para eso existe el system call *fork()*, que hace un proceso hijo que tiene una copia del stack del padre, así el proceso hijo atiende el pedido y el padre sigue escuchando nuevas peticiones. Esto es un server ***multi-process***.

```
while(1)
 { /* main accept() loop */
   addrsize = sizeof their_addr;
   new_s = accept(s, (struct sockaddr *)&their_addr, &addrsize);
   if (new_s == -1)
   {
     fprintf(stderr, "Error en accept: %s\n",gai_strerror(status));
     continue;
   }
   inet_ntop(AF_INET, &(their_addr.sin_addr),  addrstr, sizeof addrstr);
   printf("server: got connection from %s\n", addrstr);
   if (!fork())
   { /* this is the child process */
     close(s); /* child doesn't need the listener */
     if (send(new_s, "Hello, world!", 13, 0) == -1)
       perror("send");
     close(new_s);
     return 0;
   }
   close(new_s); /* parent doesn't need this */
```

Arquitectura de server (2)

- ver ex07-01-server-architecture.c
- Usar fork() en un server también implica “limpiar” la process table



- Solución: resolverlo con un signal handler
  - ver ex07-02-server-architecture-sighandler.c
- Existen otras maneras de concurrencia: ***multi-threading*** (ejemplo pthreads) https://hpc-tutorials.llnl.gov/posix/ (fuera del scope de la materia)

Blocking calls: select() and poll()

Universidad de SanAndrés

- **FD_ZERO**(), **FD_SET**(), FD_CLR(), **FD_ISSET**()

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

- readfds: test set for non-blocking read
- writefds: test set for non-blocking write
- exceptfds: test set for exceptional conditions
- ver ex08-00-select-chat.c
- poll() usa una estructura de datos y en lugar de bitmask, timeout de ms
- ambos pueden ser ineficientes cuando la cantidad de *fds* es grande (usar epoll() si desea alta eficiencia)

Recomendaciones

Universidad de SanAndrés

- Usar strace para hacer un trace de las system calls. Esto permite ver la SIGNALs que su proceso recibe (éstas a veces causan su muerte).
- Hacer error handling aunque parezca cansador (si es el mismo usar una función en vez de copiar código, no hacer lo que yo hice ;-)
- Probar las cosas por partes así cuando las junten estarán seguros que andan y que el problema está en otro lado. (yet another integration problem)
- Leer https://beej.us/guide/bgnet/
  - Usar las referencias del link de arriba (10. More References)

- Using epoch as timestamp format
- Endianness
- Framing a PDU in TCP
- Parsing fields
- Multi-process and shared-memory IPC

Vamos a medir: *sniffing*

Utilizaremos una herramienta de captura de tráfico (sniffer)

A lo largo del curso deberían familiarizarse con el wireshark, ver intercambios protocolares y poder explicarlos