Redes II - Práctica 1

Carlos Ruiz Pastor
Despacho: B-402
Correo: carlos.ruizp@uam.es

Objetivo

Diseñar un servidor Web e implementarlo en C.

* Un servidor Web con algunas limitaciones pero plenamente funcional.

Conocimientos Necesarios

- Control de versiones (Git)
- Debuggear con gdb
- C (multi-hilo, pipes)
- Leer Documentación (RFC)
- Escribir Documentación

Entregables y Fecha

- El **desarrollo** se realiza en el repositorio de **Git**.
- Los **entregables** con **estructura obligatoria**:
  - `src`
  - `srclib`
  - `include`
  - `lib`
- Es imprescindible incluir un **makefile** y **librerías propias**.
- Se valorará la calidad de los **comentarios**.

Entrega:

17 de marzo

Servidor básico

sockets, makefile, pool

Servidor HTTP

protocolo

Servidor Completo

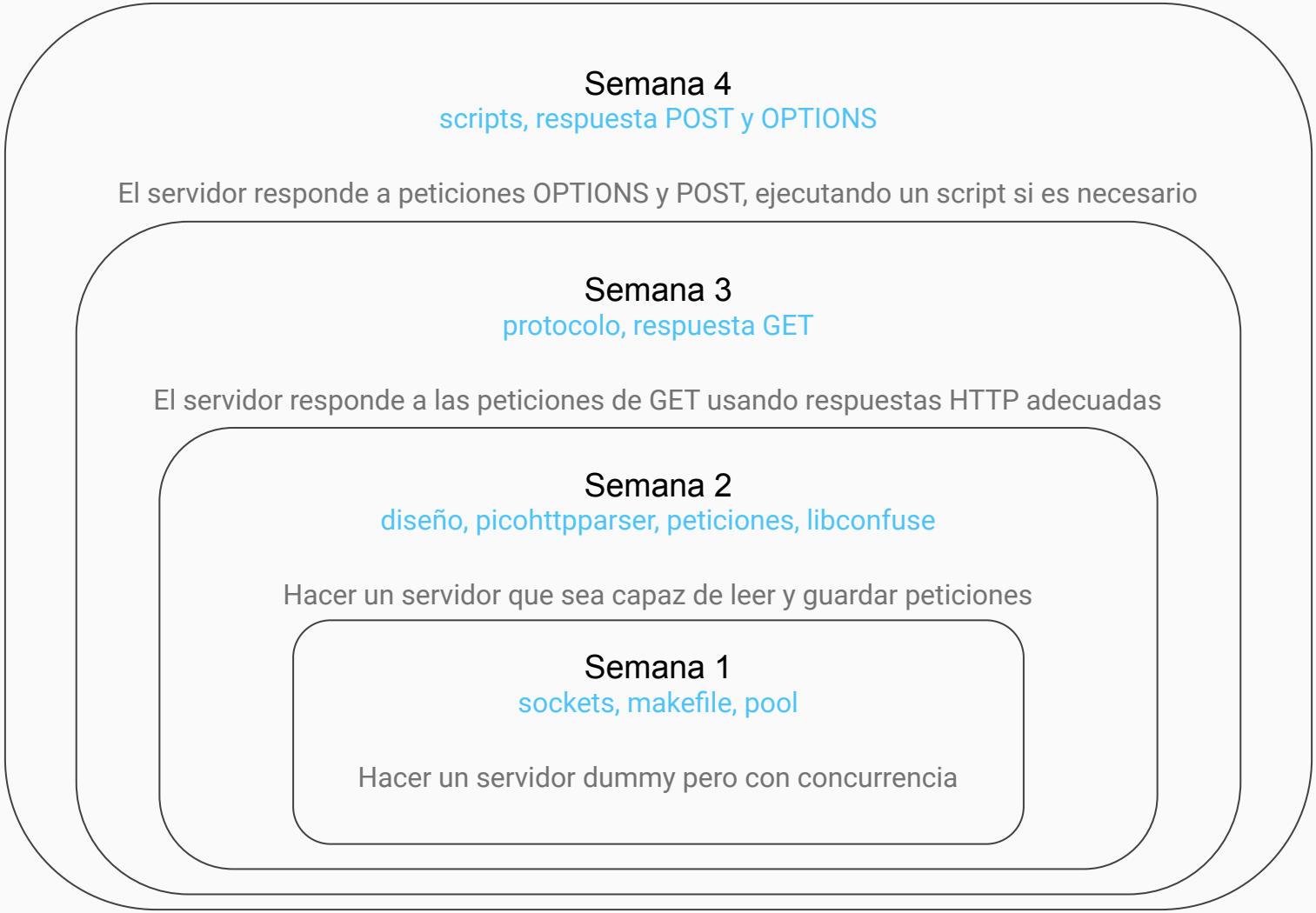
pruebas

Servidor HTTP “mudo”

diseño, picohttpparser, libconfuse

Servidor HTTP+CGI

scripts


Semana 4
scripts, respuesta POST y OPTIONS
El servidor responde a peticiones OPTIONS y POST, ejecutando un script si es necesario
Semana 3
protocolo, respuesta GET
El servidor responde a las peticiones de GET usando respuestas HTTP adecuadas
Semana 2
diseño, picohttpparser, peticiones, libconfuse
Hacer un servidor que sea capaz de leer y guardar peticiones
Semana 1
sockets, makefile, pool
Hacer un servidor dummy pero con concurrencia

Semana 1

- Leer Esquema de Comunicación, Sockets, Tipos de Servidores,
- Implementar **librería de sockets**
- Escribir **makefile** con librerías estáticas (al menos para los sockets)
- Implementar **servidor HTTP *dummy***:
  - Simple (Iterativo)
  - Multi-Hilo (mirar Thread Pool)
  - Multi-Proceso (muy similar al caso de hilos)

Semana 2

- Implementar configuración desde fichero con libconfuse
- **Comparativa** entre tipos de servidores (importante en la memoria)
  - *Iterativo* vs *Concurrente Hilos* vs *Concurrente Procesos*
  - ¿Cómo hacer peticiones en paralelo con curl?
- Leer Protocolo HTTP
- Leer RFC de HTTP
- Implementar lectura de peticiones con picohttpparser
  - ¿Es necesaria una estructura para guardar la información de las peticiones?

Semana 3

- Leer Protocolo HTTP
- Leer RFC de HTTP
- Hacer diseño a alto nivel:
  - ¿Qué funcionalidad es necesaria en todas las respuestas?
  - ¿En qué se parecen y se diferencian cada método?
  - ¿Queremos usar una estructura para la respuesta?
- Implementar respuesta a GET
  - Se pueden enviar las cabeceras en un send y el cuerpo en otro
  - El cuerpo se puede enviar en sends en un bucle

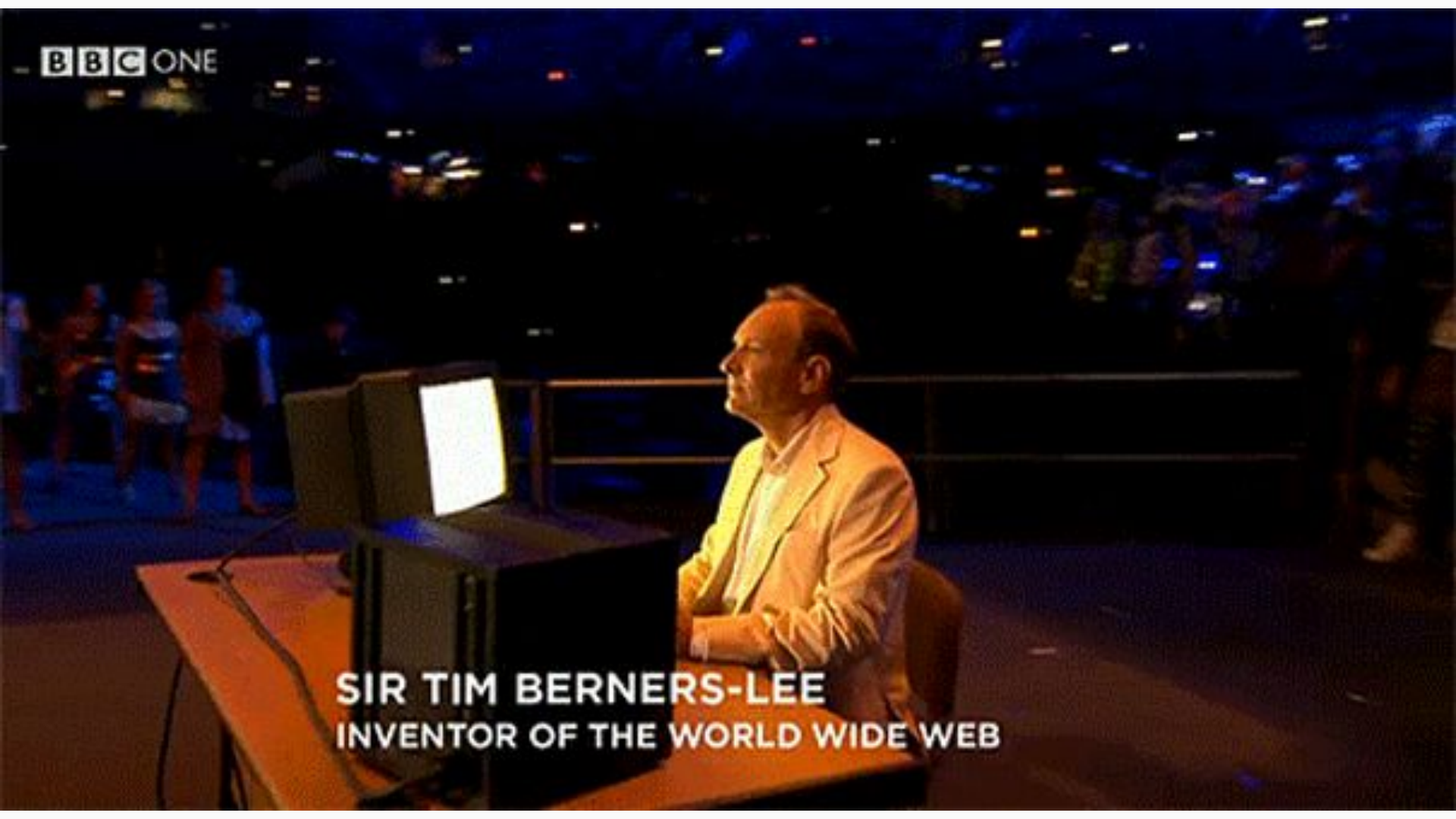
Semana 4

- Leer Protocolo HTTP
- Leer RFC de HTTP
- Implementar respuestas de POST y OPTIONS
- Añadir funcionalidad **CGI** al servidor
- Generar scripts para probar la funcionalidad CGI (opcional)

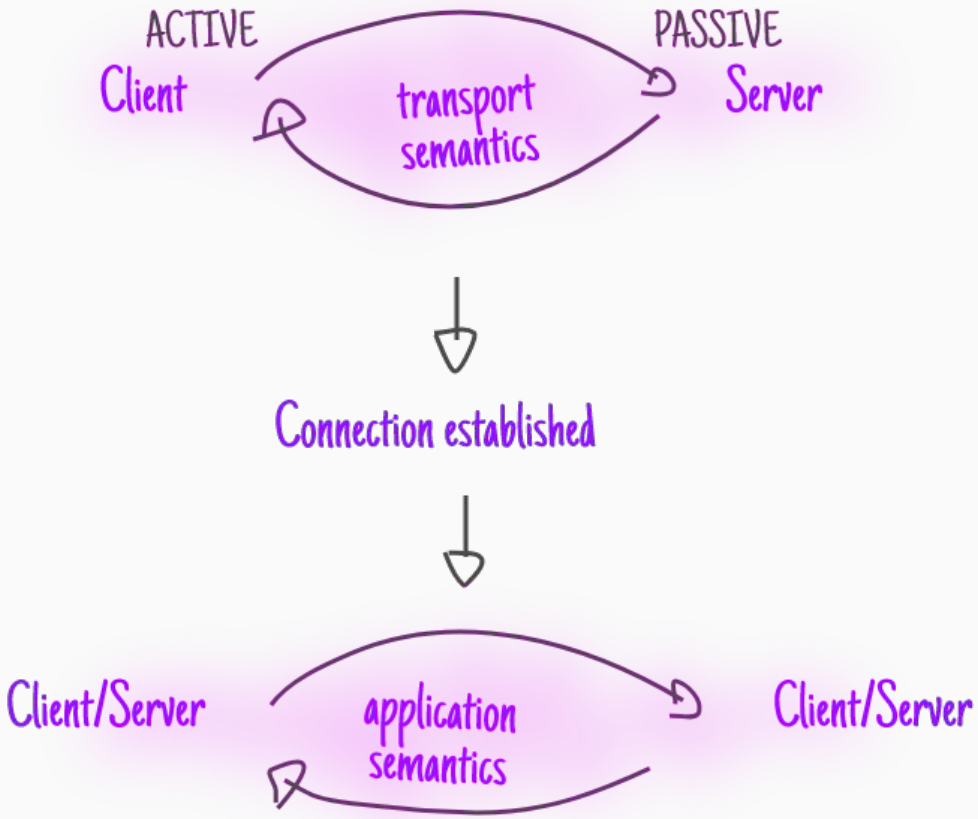
Semana 5

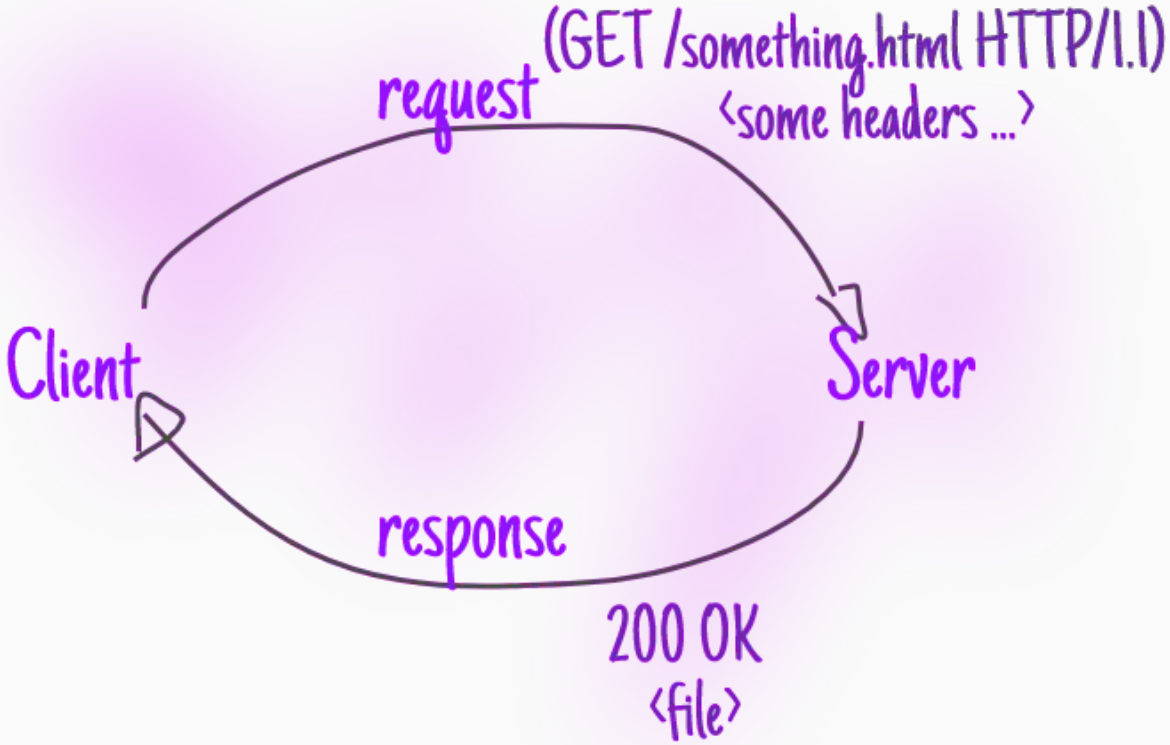
- Demonizar el servidor (leer Procesos Demonio)
- Probar servidor completo:
  - Demonio
  - Concurrente
  - Múltiples opciones de métodos/cabeceras/tipos
  - Scripts CGI
- Completar memoria

Historia HTTP


BBC ONE
SIR TIM BERNERS-LEE
INVENTOR OF THE WORLD WIDE WEB

Esquema de Comunicación


ACTIVE
Client
transport
semantics
PASSIVE
Server
Connection established
Client/Server
application
semantics
Client/Server


request
(GET /something.html HTTP/1.1)
<some headers ...>
Client
Server
response
200 OK
<file>

Sockets

UNIX Network Programming (W. Richard Stevens et al.), capítulo 4.

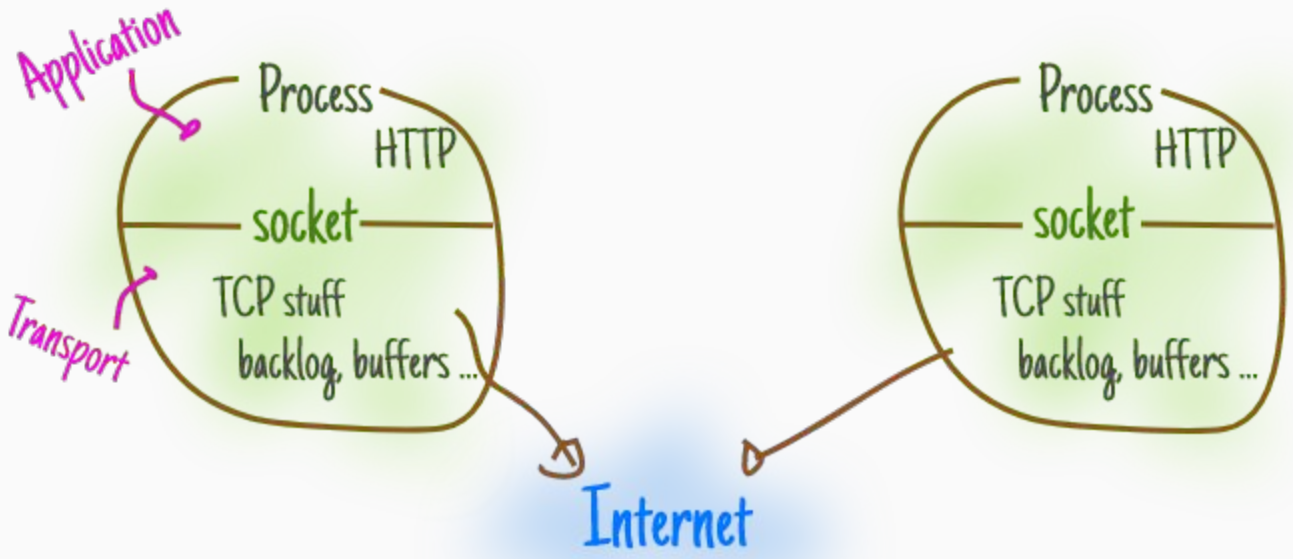
¿Qué es un socket?

Un **socket** es un tipo de *File Descriptor* que permite comunicarse con otros programas a través de una red. Encapsula las funciones de transporte.

"Everything in Unix is a file descriptor." *¿Qué es eso?*

Hay principalmente dos tipos:

- **Stream Sockets (TCP):** orientados a conexión
- **Datagram Sockets (UDP):** no orientados a conexión


Application
Process
HTTP
socket
TCP stuff
backlog, buffers ...
Transport
Process
HTTP
socket
TCP stuff
backlog, buffers ...
Internet

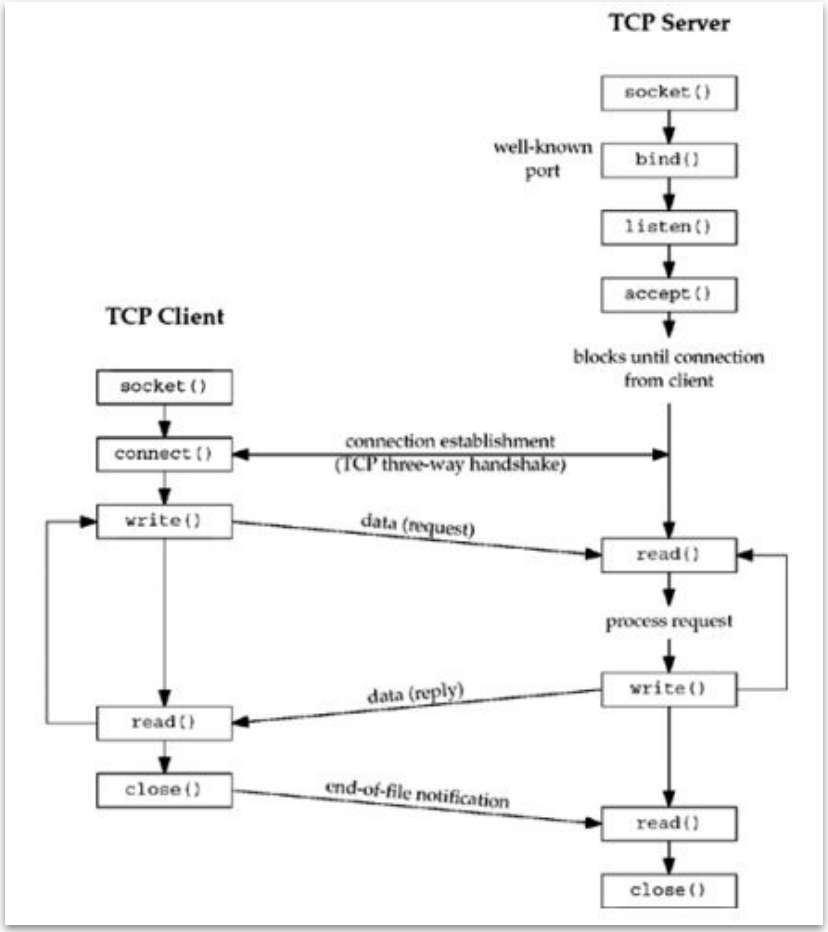
Comunicación Cliente - Servidor

Servidor

1. `socket()`: crea el socket
2. `bind()`: asocia el socket a un puerto
3. `listen()`: pone el socket en modo pasivo y crea la cola de clientes
4. `accept()`: establece la conexión con un cliente cuando recibe una petición
5. `send()/recv()`

Cliente

1. `socket()`: crea el socket
2. `connect()`: conecta el socket a un servidor
3. `send()/recv()`



UNIX Network Programming (W. Richard Stevens et al.), Figura 4.1.

Tipos de Servidores

UNIX Network Programming (W. Richard Stevens et al.), capítulo 30.

Tipos de Servidores

UNIX Network Programming with TCP/IP (W. Richard Stevens et al.), Sección 30.13

0. Iterative server (baseline measurement; no process control)
1. Concurrent server, one fork per client
2. Preforked, with each child calling accept
3. Preforked, with file locking to protect accept
4. Preforked, with thread mutex locking to protect accept
5. Preforked, with parent passing socket descriptor to child
6. Concurrent server, create one thread per client request
7. Prethreaded with mutex locking to protect accept
8. Prethreaded with main thread calling accept

Tipos de Servidores

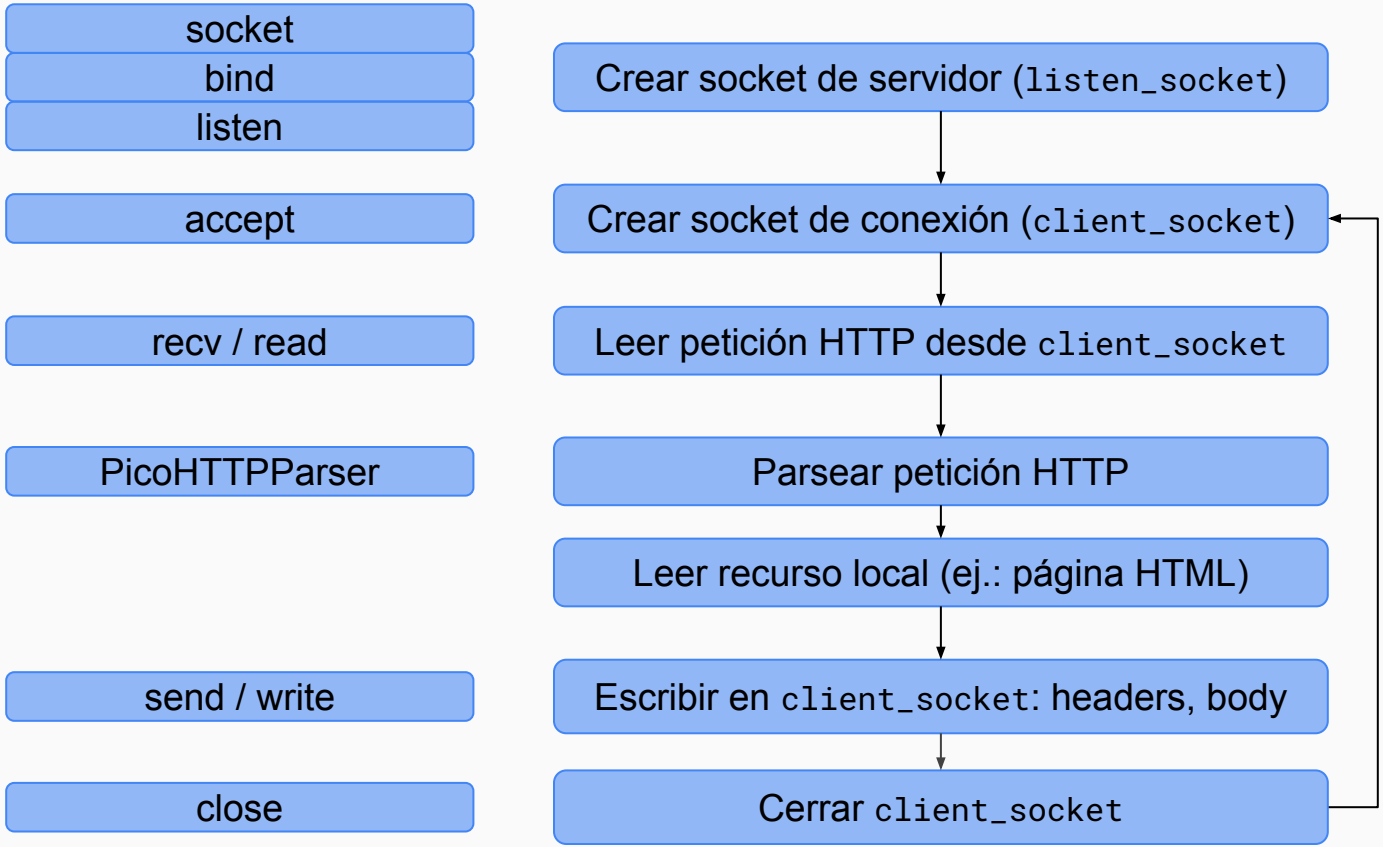
Actuales

- Servidor Iterativo
- Servidor Concurrente
  - Proceso/Hilo por petición
  - Pool de hilos / Procesos (*Apache*)
    - Estático (Esta práctica)
    - Dinámico
  - Asíncrono (*NGINX*)
  - Servidor Iterativo + Balanceador de Carga (*Gunicorn*)

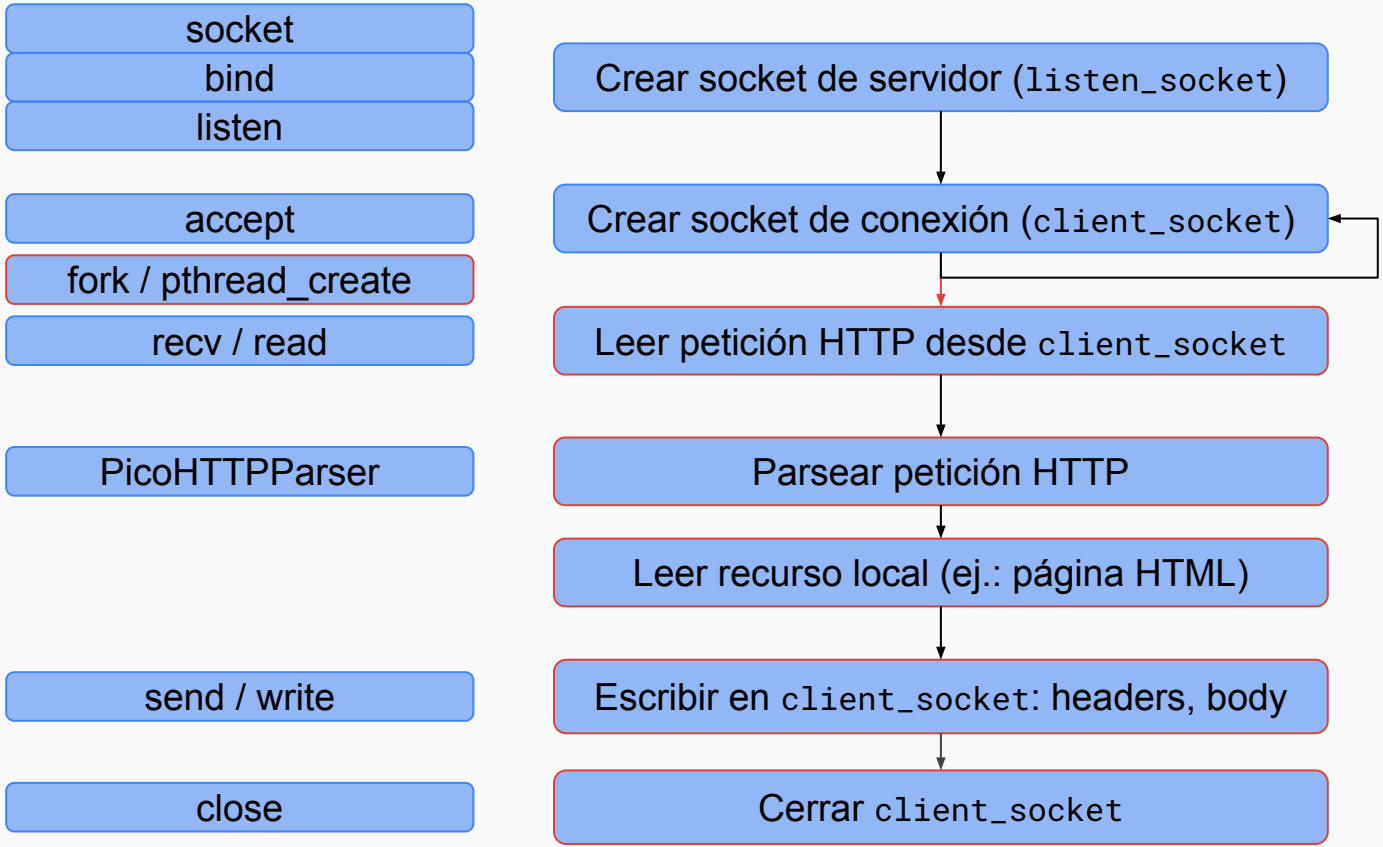
Hay que discutir en la memoria qué ventajas y desventajas tiene cada uno de estos paradigmas.

Para entender mejor el servidor asíncrono mirar Slow Poetry and the Apocalypse

Servidor HTTP Iterativo



Servidor HTTP Concurrente Simple



Servidor HTTP Concurrente con Pool de Procesos/Hilos

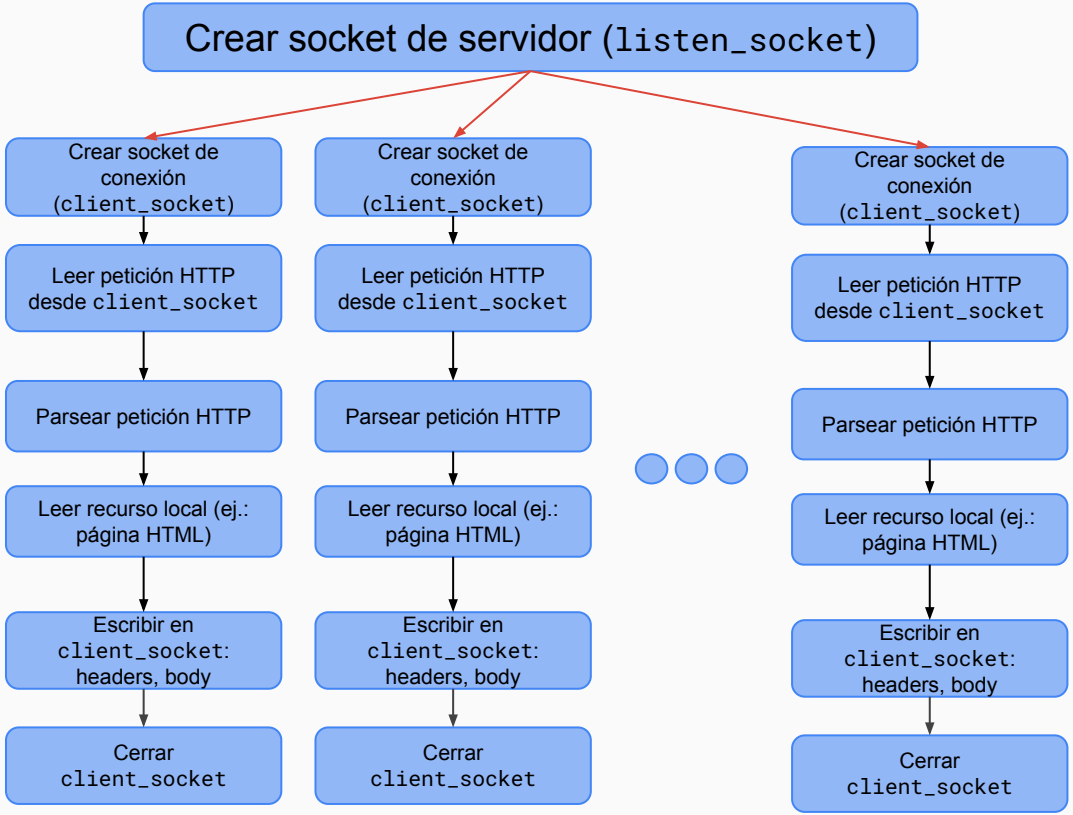
socket
bind
listen
fork / pthread_create
accept

recv / read

PicoHTTPParser

send / write

close

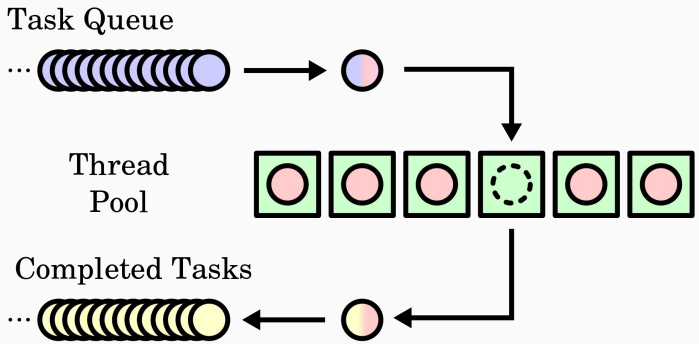


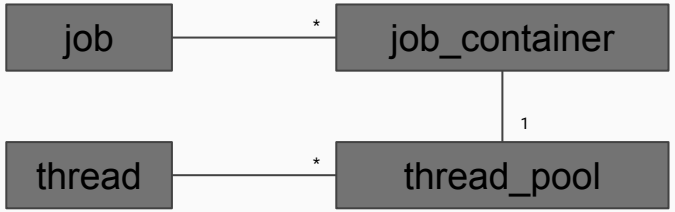
Thread Pool

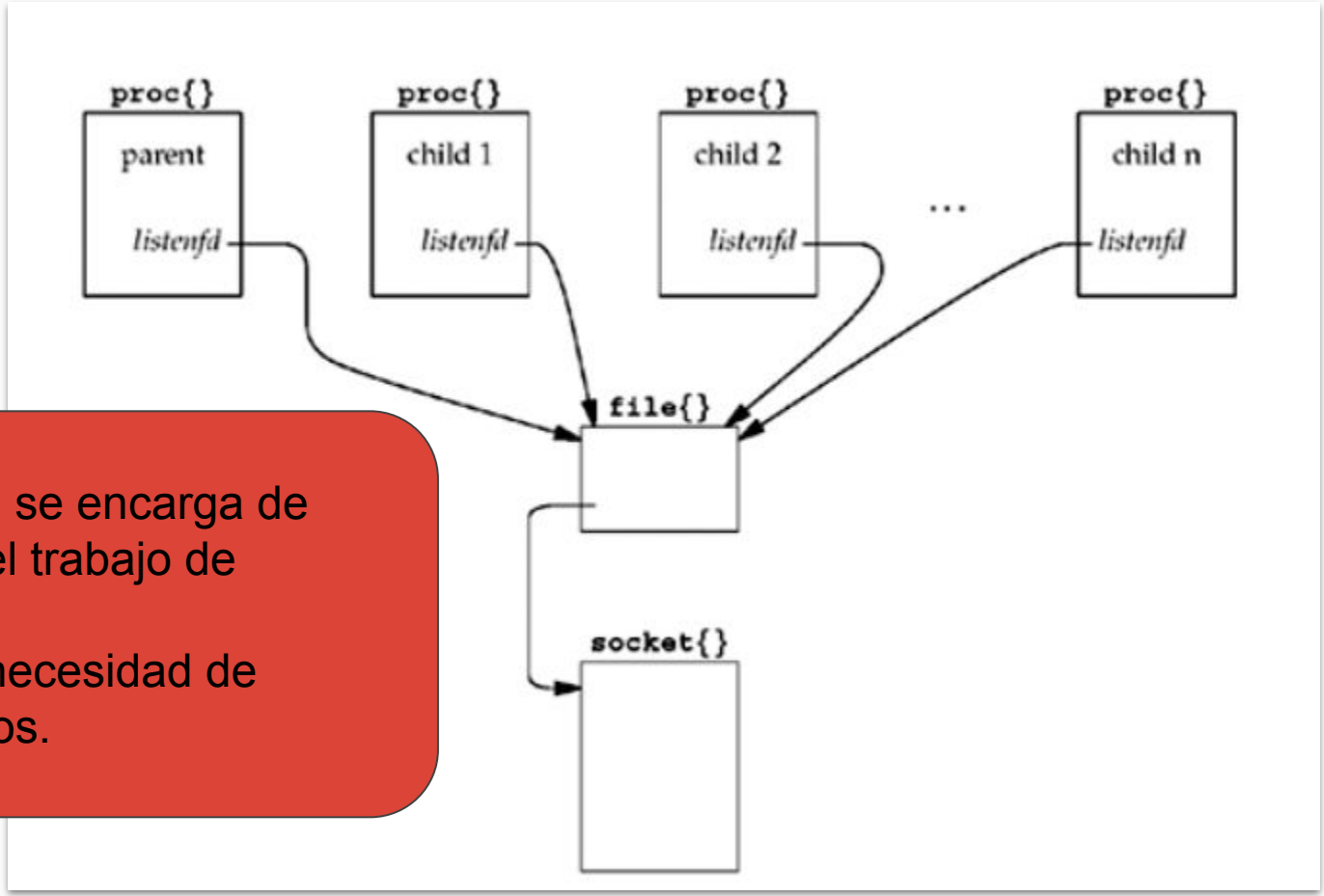
Thread Pool Típico

Requisitos:

- Dejar los hilos en un estado de espera (semáforos).
- Un contenedor de hilos (cola).
- Un contenedor de trabajo (cola).
- Una interfaz abstracta para repartir trabajo a los hilos (semáforos, condiciones).
- Una interfaz abstracta para realizar el trabajo.







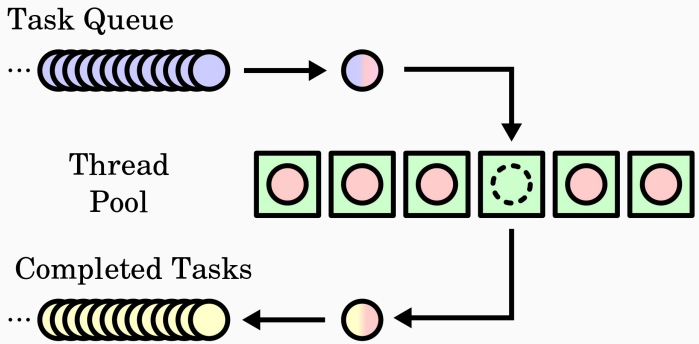
El kernel se encarga de repartir el trabajo de `accept`.
No hay necesidad de semáforos.

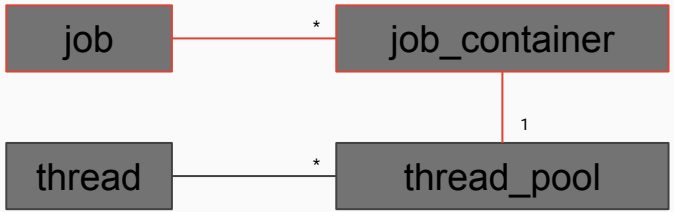
UNIX Network Programming (W. Richard Stevens et al.), Figura 30.13

Thread Pool en el servidor

Requisitos:

- Dejar los hilos en un estado de espera (**accept / mutex**).
- Un contenedor de hilos (**tamaño fijo: array**).
- Un contenedor de trabajo (**accept**).
- Una interfaz abstracta para repartir trabajo a los hilos (**accept**).
- Una interfaz abstracta para realizar el trabajo (la función de trabajo)





Pool Thread


pool_thread
num_threads
num_working
stop
shared_mutex
threads
work_function
Una variable de este tipo debería ser el argumento que se pasa en pthread_create
pool_thread.c
pool_th_ini()
th_main() / work_function()
pool_th_destroy()

Cancelar hilos

En los hilos-cliente es posible que sólo queramos que sean cancelados cuando están esperando en el `accept`. De esta manera acabaremos de responder a la petición de un cliente antes de terminar. Podemos hacerlo así:

```
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE)

while(1){

    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE)

    accept

    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE)

    ...}
```

Después podemos cancelar el hilo con `pthread_cancel` (NO con `pthread_kill`).

Liberar recursos en hilos

- No hace falta liberar los hilos-cliente en el hilo-principal si los recursos no liberados se marcan en valgrind como non-reachable:
  - Un leak es cuando dejamos memoria sin liberar (definición imprecisa)
  - Un leak es cuando dejamos memoria sin liberar y perdemos la referencia a dicha memoria (+ preciso)

  [c - Still Reachable Leak detected by Valgrind - Stack Overflow](https://stackoverflow.com/questions/3840582/still-reachable-leak-detected-by-valgrind)
- De todas maneras, si se quieren liberar recursos del hilo cuando el hilo-principal haga pthread_cancel, lo mejor sería usar la función `pthread_cleanup_push` (y `pthread_cleanup_poll`) que modifica la pila de llamadas y hace que una función (la de liberar recursos) se ejecute en el caso de la cancelación del thread.

Dejar el hilo principal en pausa

Dejar el hilo principal esperando con un while(1); es una mala práctica porque malgasta los recursos de la CPU. Una buena solución sería usar sigsuspend:

```
sigemptyset() // inicializo un set de señales vacío

sigadd(señal) // añado al set una señal

pthread_sigmask(SIG_BLOCK, new_mask, &old_mask) // bloqueo la señal para que los hilos-cliente la ignoren

sigaction(manejador_vacio) // hace falta un manejador vacío para desbloquear el hilo-principal al recibir la señal

pthread_create // creo los hilos

pthread_sigsuspend(old_mask) // Espero a recibir la señal pero con la máscara que no bloquea la señal. Mejor que pause!

pthread_cancel // cancelo los hilos-cliente, (puedo liberar los recursos en cada hilo con pthread_cleanup_push)

pthread_join // espero a que acaben, si hago exit antes de que terminen los hilos-cliente se cerrarán todos

exit
```

Procesos Demonio

UNIX Network Programming (W. Richard Stevens et al.), capítulo 13.

¿Qué es un proceso demonio?

Un demonio es un proceso que corre en el *background* y que no está asociado a ninguna terminal (*controlling terminal*).

Como no tiene asociada ninguna terminal no puede usar `stdout` o `stderr`. Lo habitual es utilizar la función `syslog` para loguear mensajes.

¿Cómo demonizar un proceso?
Método Clásico

1. `Fork`, haciendo que el padre termine.
2. Empezar una nueva sesión para el hijo llamando a `setsid`.
3. `Fork` de nuevo, haciendo que el padre termine.
4. Cambiar el directorio de trabajo a una localización segura (‘/’). (*opcional*)
5. Poner la `umask` a cero (elegir los permisos de los ficheros creados).
6. Redirigir `stdin`, `stdout` y `stderr` a `/dev/null`.

Leer UNIX Network Programming (W. Richard Stevens et al.), Sección 13.4.

¿Cómo demonizar un proceso?
Método Moderno

1. Implementar un ejecutable (cualquiera)
2. Crear un fichero nombre.service para systemd
3. Iniciar el servicio con: systemctl start nombre

Ejemplo: https://medium.com/@benmorel/creating-a-linux-service-with-systemd-611b5c8b91d6

Protocolo HTTP

Versión HTTP a implementar

- Hay que implementar la **versión HTTP 1.1**
  - No es necesario hacerlo compatible con clientes de HTTP 1.0
  - Suponemos que solo usaremos clientes que implementen HTTP 1.1
- La característica principal de esta versión es la **persistencia**
  - Las conexiones entre servidor y cliente son por defecto persistentes.
  - Esto quiere decir que el socket de conexión no se cierra hasta que se indique.
  - El final de la conexión se indica utilizando la cabecera "`Connection: close`"
- **No** es necesario **implementar** el "**pipelining**"
  - Suponemos que los clientes siempre esperan a la respuesta de una petición antes de enviar una nueva.
- **No** es necesario **implementar** el "**caching**"

Versión HTTP a implementar

Los navegadores habituales nunca envían `Connection: close`

- **Opción simple:** Esperar a que el cliente cierre la conexión, lo que se refleja en que read/recv devuelve 0 bytes.
- **Opción avanzada 1:** Añadir un timeout en el socket con setsockopt usando la opción SO_RCVTIMEO (man 7 socket)
- **Opción avanzada 2:** Usar `poll/select` sobre el file descriptor del socket de conexión para establecer un timeout.

Métodos a implementar

- **GET**: Pide recuperar cualquier información identificada por la Request-URI. Se incluye la información de la petición en la propia Request-URI.
- **POST**: Pide al servidor que la entidad de la petición sea procesada por el recurso indicado en la Request-URI. Se incluye la información de la petición en las cabeceras.
- **OPTIONS**: Se pide información sobre las opciones de comunicación disponibles en un cierto recurso identificado por la Request-URI.

Códigos de Respuesta a implementar

- **200 OK**: El procesamiento de la petición fue correcto.
- **400 Bad Request**: El servidor no pudo interpretar la petición correctamente, probablemente debido a un error de sintaxis.
- **404 Not Found**: El servidor no ha podido encontrar el recurso solicitado.

* Se pueden implementar códigos adicionales como ejercicio optativo para obtener una mayor puntuación final.

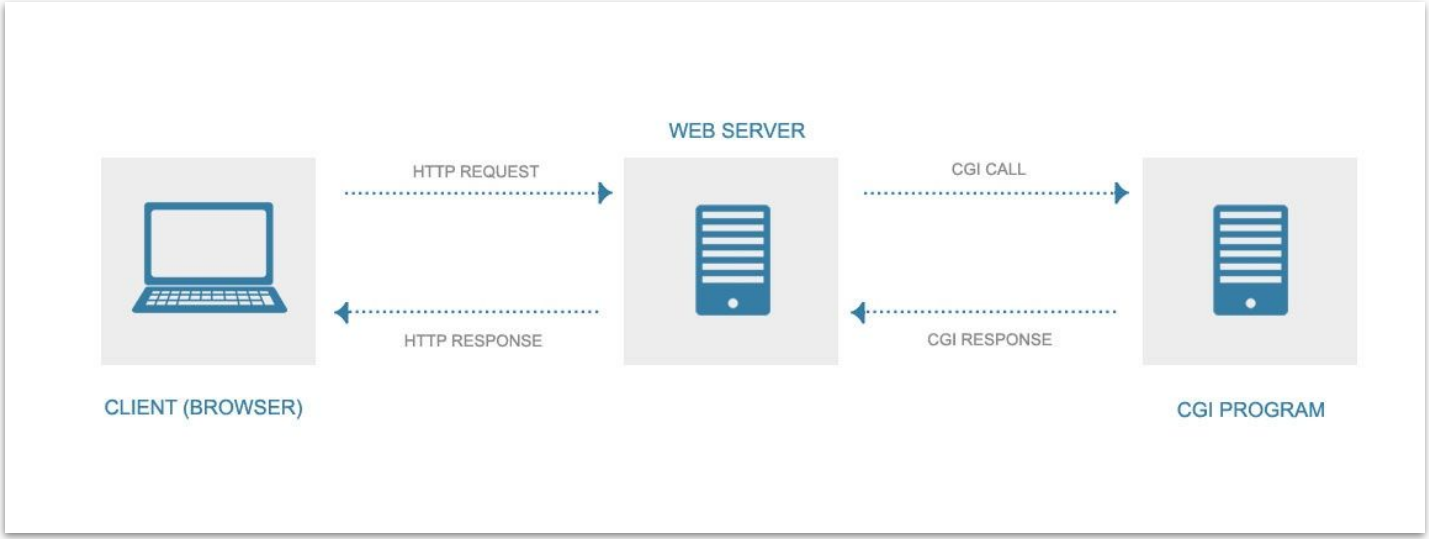
Protocolo Resumen

- **Protocolo HTTP 1.1 reducido**
- **Métodos mínimos**:
  - POST
  - GET
  - OPTIONS
- **Cabeceras mínimas**:
  - Date
  - Server
  - Last-Modified
  - Content-Length
  - Content-Type
- **Códigos de respuesta mínimos**:
  - 200 OK
  - 400 Bad Request
  - 404 Not Found
- **Tipos mínimos**:
  - text/plain: .txt
  - text/html: .html, .htm
  - image/gif: .gif
  - image/jpeg: .jpeg, .jpg
  - video/mpeg: .mpeg, .mpg
  - application/msword: .doc, .docx
  - application/pdf: .pdf

CGI Scripts

¿Qué son los CGI scripts?

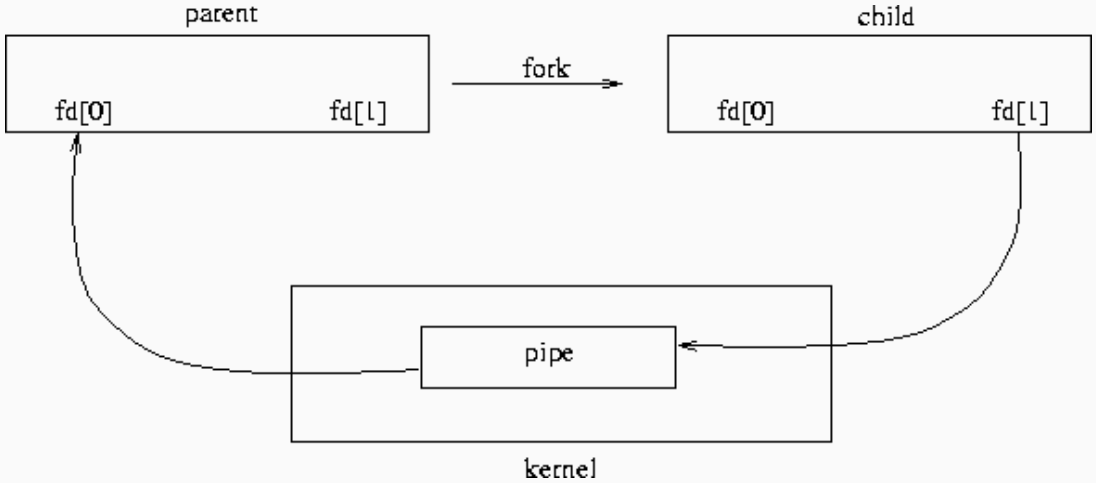
- CGI son las siglas de Common Gateway Interface
- Es una especificación que sirve para crear páginas web dinámicas
- El servidor CGI llama a otra aplicación para generar la respuesta
- Se envía la respuesta de la aplicación de vuelta al cliente usando HTTP



¿Cómo lo podemos hacer?

```
FILE *popen(const char *command, const char *type);
```

```
DESCRIPTION
       The  popen()  function opens a process by creating a pipe, forking, and
       invoking the shell.  Since a pipe is by definition unidirectional,  the
       type  argument  may  specify  only  reading  or  writing, not both; the
       resulting stream is correspondingly read-only or write-only.
```

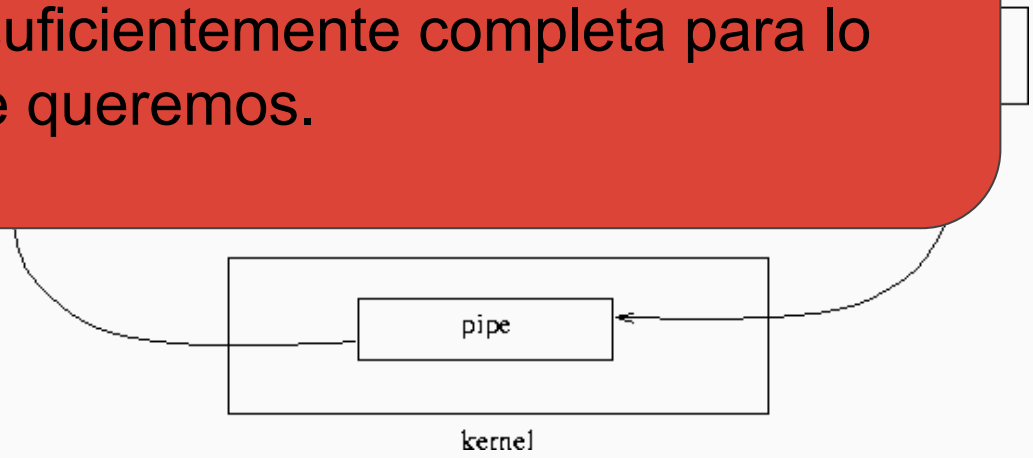


¿Cómo lo podemos hacer?

```
FILE *popen(const char *command, const char *type);
```

```
DESCRIPTION
       Th                                                    forking, and
       i                                                     ctional,  the
       t                                                     not both; the
       r                                                     y.
```

NO permite enviar datos por STDIN.
Es una opción más sencilla pero no es lo suficientemente completa para lo que queremos.



¿Cómo lo podemos hacer?

```
pid_t    pid;
int        child_in[2];
int        child_out[2];

pid = fork();
if (pid != 0) {
    // Father
    // Close the child_in pipe for reading
    close(child_in[READ]);
    // Close the child_out pipe for writing
    close(child_out[WRITE]);
    // Write to the stdin of the process if we have
something to send
    if (stdin)
        write(child_in[WRITE], stdin, strlen(stdin));
    // Close the stdin so the child does not lock
    close(child_in[WRITE]);

    // Wait for the child to die
    wait(NULL);

    return child_out[READ];
}
```

```
// Child
// Close the child_out pipe for reading
close(child_out[READ]);
// Close the child_in pipe for writing
close(child_in[WRITE]);
// child_in to stdin
dup2(child_in[READ], STDIN_FILENO);
// child_out to stdout
dup2(child_out[WRITE], STDOUT_FILENO);

return execlp(command, command, arg, (char *) NULL);
```

Para añadir un TimeOut

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

- No es necesario hacer el TimeOut pero es opcional para obtener una mayor puntuación.
- Los scripts pueden hacerse en cualquier lenguaje
- Se recomienda hacerlos en:
  - C
  - Python

```
int
main(void)
{
    fd_set rfds;
    struct timeval tv;
    int retval;

    /* Watch stdin (fd 0) to see when it has input. */

    FD_ZERO(&rfds);
    FD_SET(0, &rfds);

    /* Wait up to five seconds. */

    tv.tv_sec = 5;
    tv.tv_usec = 0;

    retval = select(1, &rfds, NULL, NULL, &tv);
    /* Don't rely on the value of tv now! */

    if (retval == -1)
        perror("select()");
    else if (retval)
        printf("Data is available now.\n");
        /* FD_ISSET(0, &rfds) will be true. */
    else
        printf("No data within five seconds.\n");

    exit(EXIT_SUCCESS);
}
```

Conclusiones

Consejos

- **No** programar **cliente**. Usar un navegador/`curl` como cliente.
- Para ello, desde el principio hacer que el servidor devuelva:

  `"HTTP/1.1 200 OK\r\n\n"` y, si es posible, una página de prueba.

- Dedicar tiempo al **diseño**.
- **Leer referencias.**
- Usar los **recursos de Internet**.

Un consejo útil

Para reusar el socket, y no esperar a que lo cierre linux, se puede añadir la opción SO_REUSEADDR al socket usando la función setsockopt (mirar man 7 tcp).

Recursos

- **[Beej's Guide to Network Programming Using Internet Sockets]** (https://beej.us/guide/bgnet/html/index-wide.html)
- **[HTTP Server in C \| Dev Notes]** (https://dev-notes.eu/2018/06/http-server-in-c/)
- **[TCP Server-Client implementation in C - GeeksforGeeks]** (https://www.geeksforgeeks.org/tcp-server-client-implementation-in-c/)
- **[HTTP RFC]** https://tools.ietf.org/html/rfc7231
- **[HTTP Made Really Simple]** https://www.jmarshall.com/easy/http/#whatis
- **[HTTP 1.0 vs 1.1]** https://stackoverflow.com/questions/246859/http-1-0-vs-1-1

Sobre la entrega

- **README** con las instrucciones básicas para compilar y ejecutar.
  - En el enunciado pone que no se debe subir al gitlab nada que no sea código, ya que el espacio es limitado. Decidme también si habéis cambiado las rutas de las imágenes, scripts, etc.
- En la **memoria**, lo más importante es:
  - **Introducción**: describir cómo se va a estructurar la memoria
  - **Decisiones de diseño**:
    - Tipo de servidor elegido, comparativa frente a otras alternativas
      1. comparativa cuantitativa con los servidores implementados: *iterativo* vs *concurrente hilos* vs *concurrente procesos*
      2. comparativa cualitativa/filosófica con otros como el asíncrono
    - Esquemas de diseño
    - Escribir qué se incluye en las librerías y el porqué
    - Cualquier otra decisión de diseño relevante y *funcionalidad extra/opcional* que hayáis implementado
  - **Esquema/Resumen a alto nivel de los módulos**: De qué se encargan, cómo se relacionan con los otros módulos. No una repetición de las cabeceras de las funciones, sino algo a alto nivel.
  - **Conclusiones**

*** Sé que la memoria es tediosa de hacer pero es especialmente útil para mí. Es más fácil leer un texto que leer código ajeno.**

Evaluación

- Uso de GIT
- Estructura de código / Diseño / Comentarios
- Documentación + Diagramas Diseño
- Funcionalidad HTTP: GET, POST, OPTIONS + RESPUESTAS + TIPOS
- Configuración desde fichero
- Logging
- Concurrencia (comparativa pool hilos vs pool procesos vs secuencial)
- Extras: códigos de error, timeout socket, timeout script, otros (indicar)
- **Examen: apto o NO apto**