Tema 2: Intercomunicación de procesos

Sistemas Distribuidos

2022-2023

Sockets: el bajo nivel

Paso de mensajes (repaso)

Paradigma fundamental. Subyace a todo.

- Primitivas básicas:
  - Enviar (`send`)
  - Recibir (`recv`)
- Modelos
  - Orientado a conexión (metáfora línea telefónica)
  - Sin conexión (metáfora sistema de correos)

Socket

- Abstracción software
- Representa un extremo de una conexión, identificado por
  - IP
  - Puerto
- Permite ignorar los detalles de las capas de transporte y red

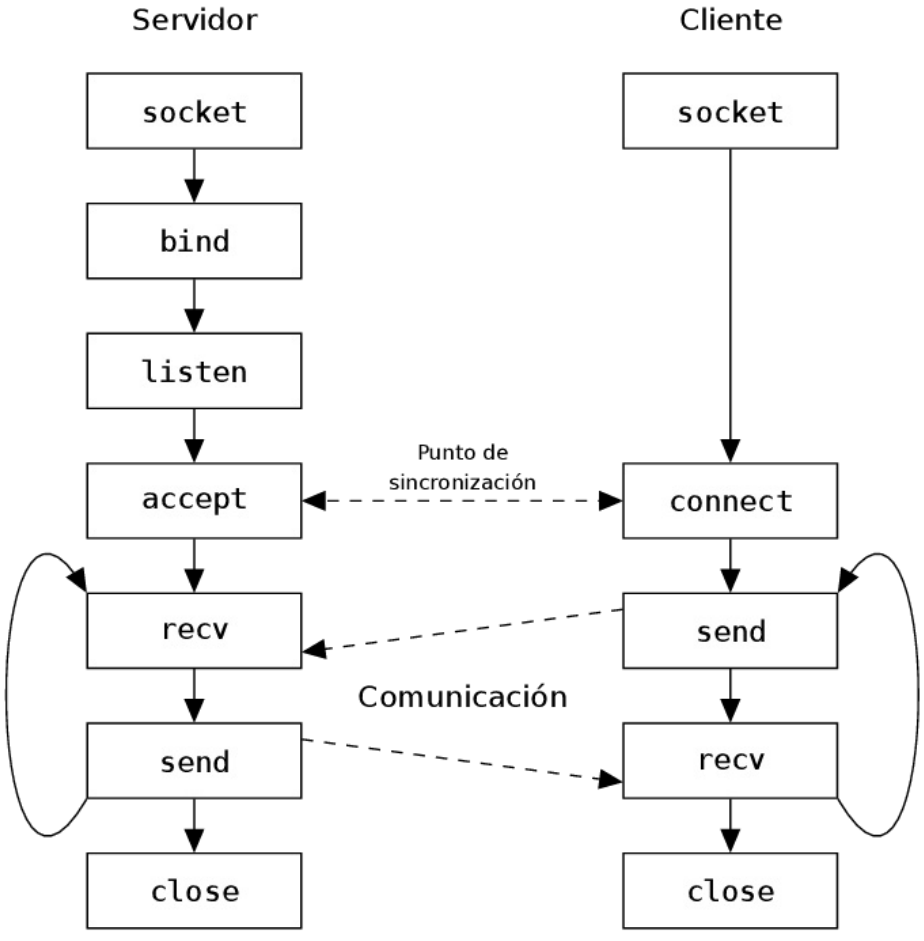
Proporciona diferentes protocolos:

- Orientado a conexión: TCP (*Transmission Control Protocol*)
- Sin conexión: UDP (*User Datagram Protocol*)

Pasos en la comunicación TCP

1. Dos procesos quieren comunicarse
2. El servidor crea un socket de escucha
3. El cliente crea un socket de datos
4. El cliente intenta conexión
5. El servidor acepta la conexión
   - Se crea un nuevo socket de datos en el servidor
   - Se establece un canal por el que pueden dialogar
6. El socket de escucha queda libre.
   Otros procesos pueden intentar conexión.
7. Cuando termina el diálogo, los sockets de datos se destruyen.

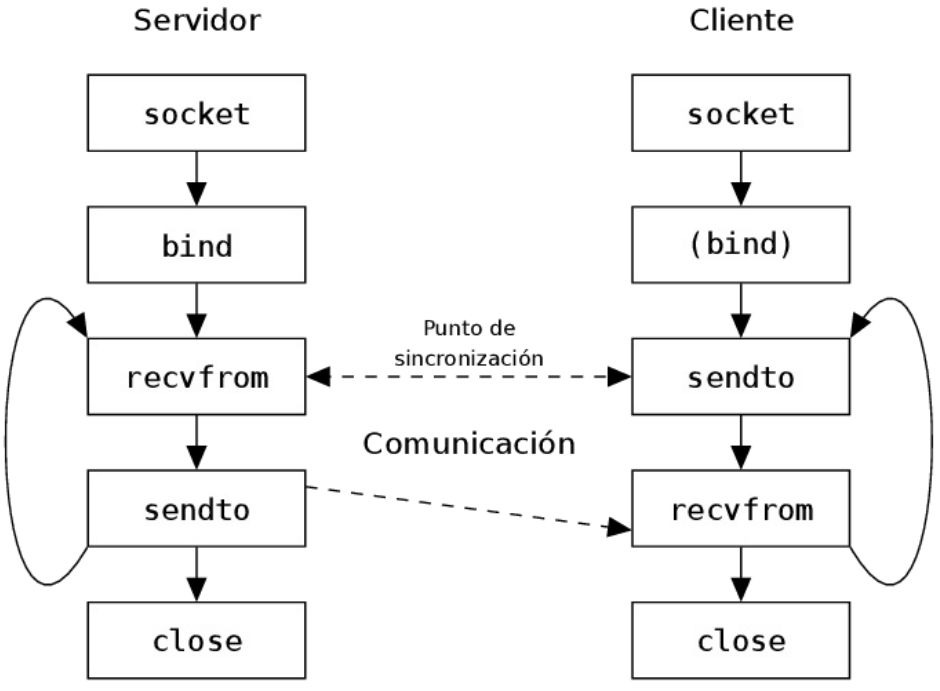
Funciones de la API para TCP



Pasos en la comunicación UDP

1. Dos procesos quieren comunicarse
2. El servidor crea un socket de datos
3. El cliente crea un socket de datos
4. El servidor espera la llegada de un datagrama
5. El cliente envía un datagrama
6. El cliente espera datagrama de respuesta
7. El servidor envía datagrama de respuesta
8. Cuando el cliente termina, destruye su socket
9. Cuando el servidor no espera más clientes, destruye su socket

Funciones de la API para UDP



Ejemplo: protocolo HOLA (TCP)

Especificación

1. El servidor espera en un socket de escucha
2. Cuando recibe conexión
    - Emite un saludo por ella
    - Cierra la conexión
3. Vuelve al paso 1

Ejemplo TCP (parte servidor)

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
int main(int argc, char* argv[]) {
    int sock_pasivo, sock_datos;
    struct sockaddr_in d_local;
    char *mensaje = "Hola, cliente";

    sock_pasivo = socket(PF_INET, SOCK_STREAM, 0);
    d_local.sin_family = AF_INET;
    d_local.sin_addr.s_addr = htonl(INADDR_ANY);
    d_local.sin_port = htons(7890);
    bind(sock_pasivo, (struct sockaddr *)&d_local, sizeof(d_local));
    listen(sock_pasivo, SOMAXCONN);

    while (1) {  // Bucle infinito de atención a clientes
        sock_datos = accept(sock_pasivo, 0, 0);
        send(sock_datos, mensaje, strlen(mensaje), 0);
        close(sock_datos);
    }
}
```

Ejemplo TCP (parte cliente)

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <stdio.h>
int main(int argc, char* argv[]) {
    int sock_datos;
    struct sockaddr_in d_serv;
    char buffer[200];
    int recibidos;

    sock_datos = socket(PF_INET, SOCK_STREAM, 0);
    d_serv.sin_family = AF_INET;
    d_serv.sin_addr.s_addr = inet_addr("127.0.0.1");
    d_serv.sin_port = htons(7890);
    connect(sock_datos, (struct sockaddr *)&d_serv, sizeof(d_serv));
    recibidos = recv(sock_datos, buffer, 200, 0);
    printf("Recibidos %d bytes\n", recibidos);
    buffer[recibidos] = 0;  // Añadir terminador
    printf("Mensaje: %s\n", buffer);
    close(sock_datos);
    return 0;
}
```

Ejemplo: protocolo HOLA (UDP)

Especificación

1. El servidor espera un datagrama
2. Cuando recibe uno
    - Ignora su contenido (puede estar vacío)
    - Envía al remitente otro datagrama con el saludo
3. Vuelve al paso 1.

Ejemplo UDP (parte servidor)

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <arpa/inet.h>
int main(int argc, char* argv[]) {
    int sock_dat, recibidos;
    struct sockaddr_in d_local, d_cliente;
    socklen_t ldir = sizeof(d_cliente);
    char *mensaje = "Hola, cliente";
    char buffer[50];

    sock_dat = socket(PF_INET, SOCK_DGRAM, 0);
    d_local.sin_family = AF_INET;
    d_local.sin_addr.s_addr = htonl(INADDR_ANY);
    d_local.sin_port = htons(7890);
    bind(sock_dat, (struct sockaddr *)&d_local, sizeof(d_local));
    while (1) {  // Bucle infinito de atención a clientes
      recibidos = recvfrom(sock_dat, buffer, 50, 0,
                           (struct sockaddr *) &d_cliente, &ldir);
      printf("Cliente desde %s (%d)\n",
          inet_ntoa(d_cliente.sin_addr), ntohs(d_cliente.sin_port));
      sendto(sock_dat, mensaje, strlen(mensaje), 0,
                           (struct sockaddr *) &d_cliente, ldir);
    }
}
```

Ejemplo UDP (parte cliente)

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <arpa/inet.h>
int main(int argc, char* argv[]) {
    int sock_dat, recibidos;
    struct sockaddr_in d_serv;
    socklen_t ldir = sizeof(d_serv);
    char buffer[50];

    sock_dat = socket(PF_INET, SOCK_DGRAM, 0);
    d_serv.sin_family = AF_INET;
    d_serv.sin_addr.s_addr = inet_addr("127.0.0.1");
    d_serv.sin_port = htons(7890);
    // Enviar datagrama vacio
    sendto(sock_dat, buffer, 0, 0, (struct sockaddr *) &d_serv, ldir);
    recibidos = recvfrom(sock_dat, buffer, 50, 0,
                         (struct sockaddr *) &d_serv, &ldir);
    buffer[recibidos] = 0;  // Añadir terminador
    printf("Mensaje recibido: %s\n", buffer);
}
```

Sugerencias

- Añadir toda la gestión de errores.
- Modificar el servidor `HOLA` para que
  - En lugar del mensaje `"Hola cliente"`
  - Envíe un mensaje que contenga la IP del cliente
    (Extraída de la conexión o del remite del datagrama)

Este protocolo podría ser útil para averiguar cuál es la IP que el servidor "ve" de nuestro cliente.

Atención a múltiples clientes

Problema: Las funciones `accept()`, `read()`, `write()` son *bloqueantes.*

- Mientras `read()` no reciba datos, *bloquea* al proceso
- Eso le impide hacer `accept()` a otro cliente.

```
// Pseudocódigo de servidor genérico
Crear socket pasivo sp
bucle infinito:
    sd = accept(sp)
    bucle de atención al cliente
        read()
        write()
        (hasta que el cliente cierre)
    close(sd)
```

Soluciones

- Usar múltiples procesos (`fork()`)
- Usar múltiples hilos en un solo proceso
- Usar un solo hilo y `select()` para evitar bloqueos
- Hacer los sockets *no bloqueantes*.

Múltiples procesos

- Esta solución (tal como se muestra aquí) sólo está disponible en Unix
- Windows carece de la función `fork()`

```
#include <unistd.h>
pid_t fork(void);
```

Esta función:

- "clona" el estado del proceso en el instante de su ejecución
  - Código
  - Variables globales
  - Variables locales
- La ejecución prosigue en paralelo en el "padre" y el "hijo"
- `fork()` retorna 0 al hijo, y el *pid* del hijo al padre

Ejemplo de fork()

```c
#include <stdio.h>
#include <unistd.h>
int main() {
    int i, p;

    for (i=0; i<2; i++) {
        p = fork();
        printf("i=%d, p=%d\n", i, p);
    }
}
```

¿Qué saldrá? ¿cuántos procesos se crean?

fork(): Creación de procesos bajo demanda

- El padre crea el socket pasivo y el de datos
- El hijo se crea tras recibir el cliente
- El hijo atiende al cliente (y luego muere)
- El padre vuelve a la espera de otro cliente
- Cuidado con los *zombies*

```
// Pseudocódigo de servidor genérico
Crear socket pasivo sp
bucle infinito:
    sd = accept(sp)
    p = fork()
     if (p==0)  // Soy el hijo
       close(sp)
      bucle de atención al cliente (sd)
      exit()    // Muere al acabar el servicio
    else        // Soy el padre
      close(sd)
   // el padre vuelve al bucle infinito
```

fork(): Creación previa de procesos servidores

- Se crea el socket pasivo
- Se clona N veces el proceso (y por tanto el socket)
- Todos los procesos (idénticos) se comportan como un servidor iterativo.
- Es necesario regular el acceso al `accept()`

```
// Pseudocódigo de servidor genérico
Crear socket pasivo sp
llamar a fork() N veces
bucle infinito: (N+1 procesos)
    exclusión mutua:
       sd = accept(sp)
    fin exclusión mutua
    bucle de atención al cliente (sd)
    close(sd)
```

Un solo proceso, múltiples hilos

- Un hilo es como un *proceso ligero*.
- Todos los hilos de un mismo proceso comparten:
  - Código
  - Variables globales
  - *heap*
- Se diferencian en que cada uno tiene:
  - Su propio *PC* (flujo de ejecución)
  - Su propia pila (variables locales)
- Un hilo puede verse como la ejecución de una función *en paralelo* con otra.

> Las ideas para hacer servidores son las mismas que las ya vistas para el `fork()`

`select()`: Multiplexación de la Entrada/Salida

- `select()` puede examinar *varios descriptores simultáneamente*
- Se bloquea mientras no haya actividad en *ninguno*
- Se desbloquea cuando hay actividad en *cualquiera*
- Retorna una lista de aquellos en que hubo actividad

De este modo el servidor:

1. Prepara lista de todos en los que espera algo . Llama a `select()` y duerme . Según en cuál haya ocurrido algo, hace:
   - Si fue un socket pasivo, `accept()`
   - Si fue uno de datos, da servicio . Vuelve al paso 1

Pseudocódigo con select()

Servidor que atiende a varios clientes conectados a la vez que acepta clientes nuevos.

```
Vaciar lista de sockets de datos
Crear socket pasivo sp
bucle infinito:
  Actualizar conjunto de descriptores a vigilar con select
    (debe incluir sp más cada socket activo en la lista)
  select()   // bloqueo hasta que haya actividad
  if (hubo actividad en sp):
     sd = accept(sp)
     if (hay sitio en la lista):
        Añadir sd a la lista
     else:
        close(sd)
  else:
     para cada socket s en la lista:
        if (hubo actividad en s):
           dar servicio (s)
           if (detectado fin de transmisión en s):
              eliminar s de la lista
              close(s)
  // Volver al bucle
```