Redes de Computadores

Introducción a la programación con sockets

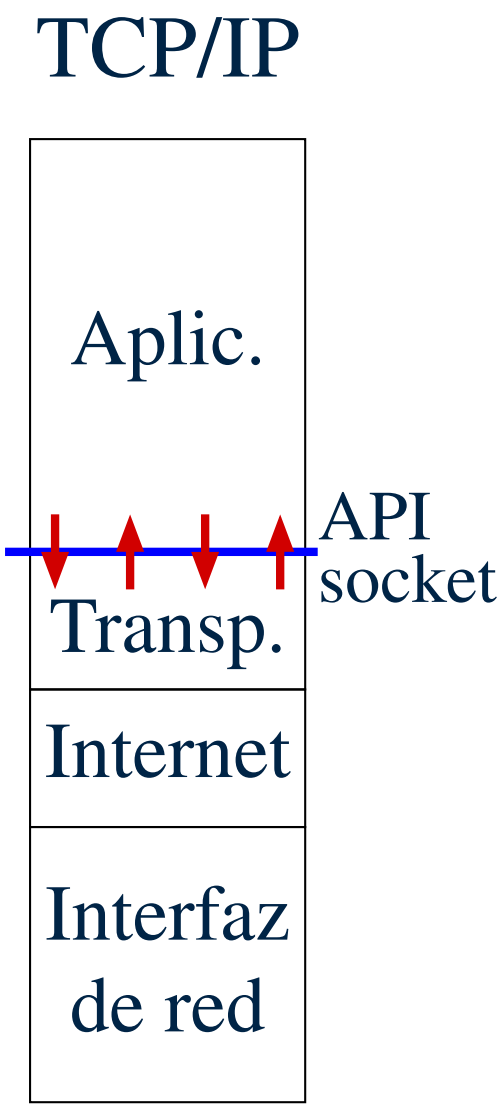
Natalia Ayuso, Juan Segarra y Jesús Alastruey

Departamento de
Informática e Ingeniería
de Sistemas

Universidad Zaragoza

Índice

1 Introducción



Aplicaciones en red: programas que se comunican mediante un protocolo de aplicación a través de una red de computadores

Application Program Interface (API) socket: interfaz de programación entre capa de aplicación y capa de transporte originalmente desarrollado para BSD (*Berkeley Software Distribution*), un derivado de Unix

socket (enchufe/conector): extremo de un canal de comunicación entre dos procesos.

2 Tipos de sockets y protocolos

familia/dominio: especifica el formato de las direcciones que se podrán dar al socket y los diferentes protocolos soportados por las comunicaciones vía los sockets de ese dominio

```
$ man socket
[...]
The domain argument specifies a communication domain;
this selects the protocol family which will be used
AF_UNIX, AF_LOCAL   Local communication              unix(7)
AF_INET             IPv4 Internet protocols         ip(7)
AF_INET6            IPv6 Internet protocols         ipv6(7)
```

```
$ man s7 ip
```

2 Tipos de sockets y protocolos (II)

Socket de tipo flujo: *stream* o TCP (`SOCK_STREAM`)

- Transmisión bidireccional continua (*stream*)
- Fiable: los datos se reciben ordenados, sin errores, sin pérdidas y sin duplicados
- Con conexión: el emisor se pone en contacto con el receptor antes de enviar datos
- Prot. capa de transporte: TCP (*Transmission Control Protocol*)

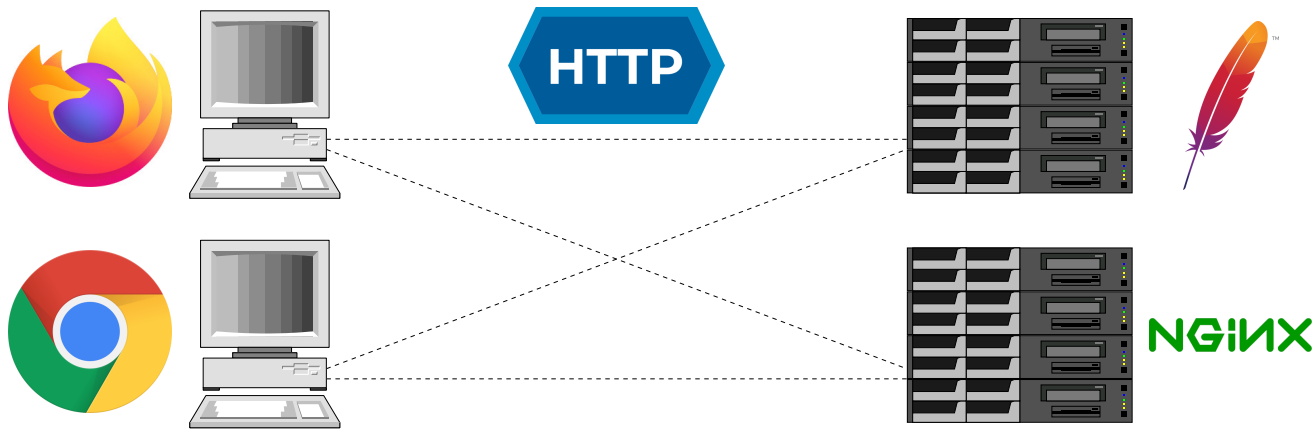
Socket de tipo datagrama: *datagram* o UDP (`SOCK_DGRAM`)

- Transmisión bidireccional de paquetes de tamaño limitado
- No fiable
- Sin conexión
- Prot. capa de transporte: UDP (*User Datagram Protocol*)

Otros (dependiendo del sistema o familia utilizado)

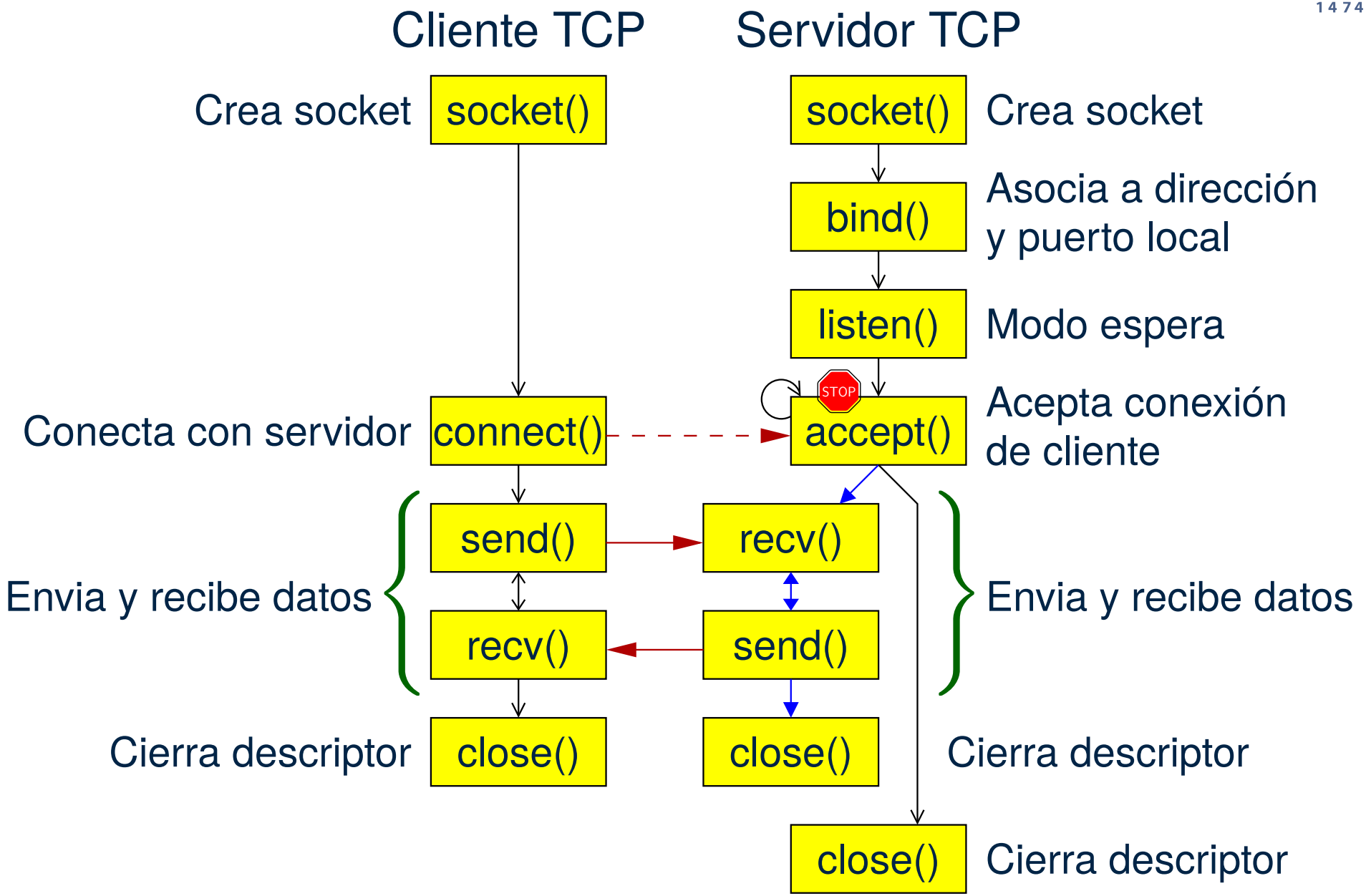
3 Modelo cliente/servidor

- Programación basada en modelo cliente/servidor
  - Cliente: proceso que inicia la comunicación para solicitar un servicio (e.g. navegador web solicitando una página)
  - Servidor: proceso que recibe y atiende solicitudes de varios clientes (e.g. servidor web enviando la página solicitada)

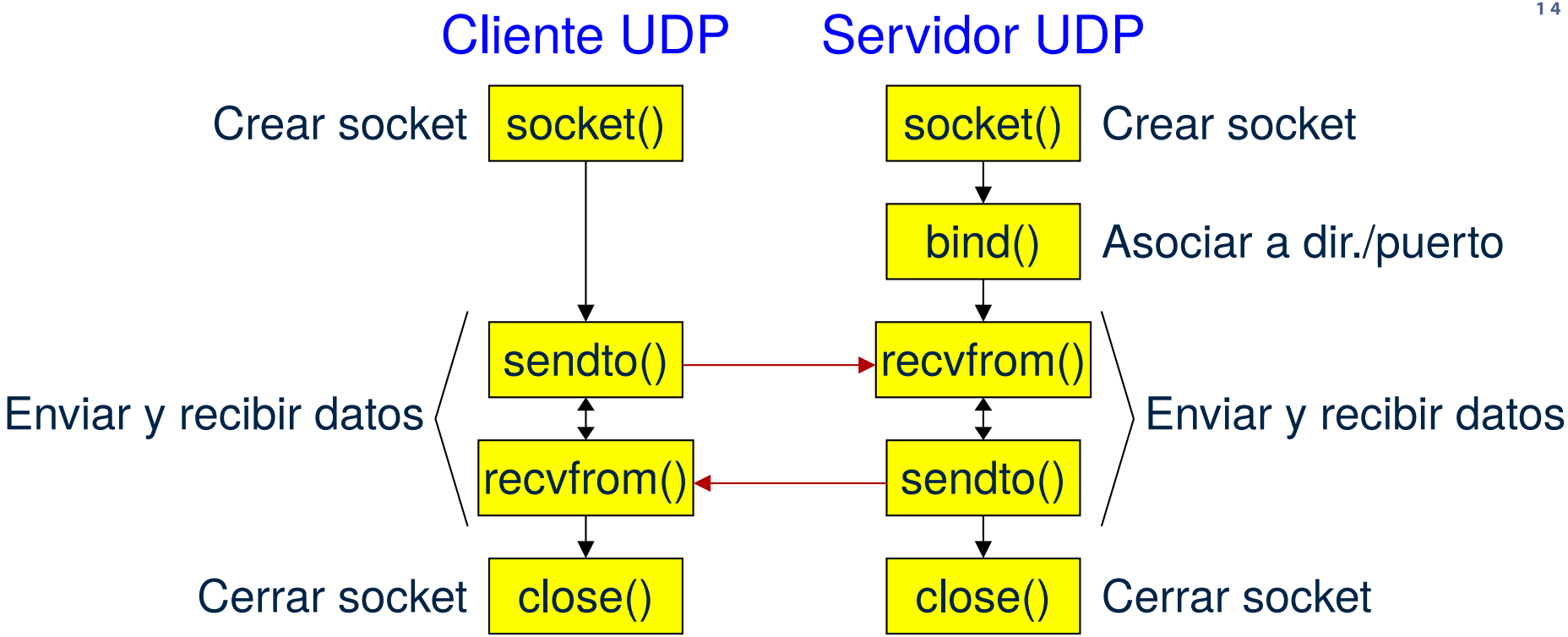


- Dependiendo del tipo de socket utilizado (TCP o UDP), el procedimiento para manejar las funciones varía ligeramente

3.1 Modelo cliente/servidor TCP



3.2 Modelo cliente/servidor UDP



- No se usan las llamadas de establecimiento de conexión (listen/connect/accept)
- Las llamadas de envío/recepción de datos (sendto/recvfrom) incluyen destinatario/remitente por no haber conexión previa

4 Llamadas al sistema

socket=socket(dominio, tipo, protocolo) crea un extremo de la comunicación

```
int fd = socket(AF_INET,SOCK_STREAM,0); //Selects TCP
```

```
socket(AF_INET,SOCK_STREAM,6); // Also selects TCP
socket(AF_INET,SOCK_STREAM,IPPROTO_TCP); // Idem
socket(AF_INET,SOCK_DGRAM,0); // Selects UDP
socket(AF_INET,SOCK_SEQPACKET,0); // Selects SCTP (not
    implemented on macOS)
socket(AF_BLUETOOTH,SOCK_STREAM,BTPROTO_RFCOMM); //
    Selects the Bluetooth RFCOMM protocol
```

4 Llamadas al sistema (II)

err=bind(socket, dir_local, long_dir) enlaza socket a la dirección y puerto locales (dir_local)

err=listen(socket, long_cola) pone socket en modo de espera y establece su límite máximo de conexiones en espera

err=connect(socket, dir_remota, long_dir) inicia una conexión en socket con dir_remota que debe estar en modo de espera

sock2=accept(socket, ⬆ dir_remota, ↕ long_dir) crea una conexión (sock2) a partir de una petición (connect) a socket (modo espera) e informa sobre el cliente (dir_remota)

núm=send(socket, mensaje, long_msj, flags) envía mensaje a través de socket

núm=recv(socket, mensaje, long_msj, flags) recibe mensaje a través de socket

4 Llamadas al sistema (III)

núm=sendto(socket, mensaje, long_msj, flags,dir_remota,long_dir)
envía mensaje a dir_remota a través de socket

núm=recvfrom(socket,msj,long_msj,flags,↑ dir_remota,↕ long_dir)
recibe mensaje e informa sobre el remitente (dir_remota)

err=shutdown(socket, sentido) cierra socket en uno o ambos sentidos

. . . . . .

núm=read(descriptor, mensaje, long_msj) lee mensaje de descriptor (socket/fichero)

núm=write(descriptor, mensaje, long_msj) escribe mensaje en descriptor (socket/fichero)

err=close(descriptor) cierra descriptor (socket/fichero)

4.1 Llamadas y funciones auxiliares

htons/htonl/ntohs/ntohl convierte (*to*) enteros cortos/largos (short/long) entre formato local y red (host y network)

getaddrinfo traduce servicios y direcciones de red

freeaddrinfo libera mem. dinámica de estructuras de dirección

inet_ntop/inet_pton analiza/crea estructuras de dirección de red

getsockopt/setsockopt obtiene/establece opciones en sockets

5 Estructuras de direcciones

- Las direcciones se guardan en un `struct addrinfo`

```
struct addrinfo {
 int      ai_flags;          //AI_PASSIVE,AI_CANONNAME,etc
 int      ai_family;         //AF_INET,AF_INET6,AF_UNSPEC
 int      ai_socktype;       //SOCK_STREAM, SOCK_DGRAM
 int      ai_protocol;       //use 0 for "any"
 size_t   ai_addrlen;        //size of ai_addr in bytes
 struct sockaddr *ai_addr;   //struct sockaddr_in or _in6
 char    *ai_canonname;      //full canonical hostname

 struct addrinfo *ai_next;   //linked list, next node
};
```

- El formato y tamaño de las direcciones (`struct sockaddr *ai_addr`) depende de la familia/dominio

5 Estructuras de direcciones (II)

- Estructura de direcciones para el protocolo IPv4 de Internet (familia `AF_INET`/`PF_INET`)

```
struct sockaddr_in {
 short          sin_family;   // AF_INET
 unsigned short sin_port;     // e.g. htons(3490)
 struct in_addr sin_addr;     // see struct in_addr below
 char           sin_zero[8]; // zero this if you want to
};
```

```
struct in_addr {
 unsigned long s_addr;        // 32bit X.X.X.X
};
```

5 Estructuras de direcciones (III)

- Estructura de direcciones para el protocolo IPv6 de Internet (familia `AF_INET6/PF_INET6`)

```
struct sockaddr_in6 {
 u_int16_t       sin6_family;    // addr family AF_INET6
 u_int16_t       sin6_port;   // port, Network Byte Order
 u_int32_t       sin6_flowinfo; // IPv6 flow information
 struct in6_addr sin6_addr;      // IPv6 address
 u_int32_t       sin6_scope_id; // Scope ID
};
```

```
struct in6_addr {
 unsigned char   s6_addr[16];   // 128bit X:X:X:X:X:X:X:X
};
```

5 Estructuras de direcciones (IV)

- Estructura de direcciones para cuando se desconoce la familia específica (reserva el tamaño suficiente)

```
struct sockaddr_storage {
    sa_family_t  ss_family;     // address family

//all this is implementation specific padding, ignore it
    char        __ss_pad1[_SS_PAD1SIZE];
    int64_t     __ss_align;
    char        __ss_pad2[_SS_PAD2SIZE];
};
```

- Las estructuras de direcciones (`sockaddr`, `sockaddr_in`, `sockaddr_in6` y `sockaddr_storage`) son intercambiables
- Cualquier función con alguna de las estructuras anteriores como parámetro sabrá tratar cualquiera de ellas

6 Cambio explícito de tipos

Ejemplo con parámetro tipo `struct sockaddr` de salida:

```
int accept(int s, struct sockaddr *addr, socklen_t
*addrlen);
```

- Si esperamos sólo direcciones IPv4:

```
struct sockaddr_in *addr_ipv4;
accept(sock,                    addr_ipv4, &len)
```

- Si esperamos sólo direcciones IPv6:

```
struct sockaddr_in6 *addr_ipv6;
accept(sock,                    addr_ipv6, &len)
```

- Si esperamos direcciones IPv4 e IPv6:

```
struct sockaddr_storage addr_ip;
accept(sock,                     addr_ip, &len)
```

Sólo lo vamos a usar sobre punteros para evitar sus *warnings*

7 Serialización de datos

- En TCP/IP no hay capa de presentación de datos
- El programador decide el formato de intercambio y debe garantizar que ambos extremos lo interpretan igual
- send/recv transmiten bytes contiguos en memoria

  Tipos básicos: ya almacenados en bytes contiguos

  Enteros: htons/htonl/ntohs/ntohl

  Estructuras de datos dispersos en memoria: programar funciones para recorrer y enviar (o recibir y generar) la estructura (listas, tablas *hash*, ¿structs?, etc.)

7 Serialización de datos (II)

Ejemplo de envío (tipo `void` no da *warnings*):

```
ssize_t send(int s, const void *msg, size_t len, int
flags);
```

- Si queremos enviar una cadena/vector:

```
char cadena[10];
send(s,        , sizeof(      ), flags);
```

- Si queremos enviar un entero largo, e.g. 20:

```
long numero;

send(s,        , sizeof(      ), flags);
```

- Si queremos enviar un `struct` con datos contiguos y correctamente formateados:

```
struct mensaje mistruct;
send(s,          , sizeof(          ),flags);
```