Network and Sockets

Slides are mainly taken from «*Operating Systems: Internals and Design Principles”, 7/E* William Stallings (Chapter 17).

Sistemi di Calcolo 2

Instructor: Riccardo Lazzeretti

Special thanks to: Daniele Cono D’Elia, Leonardo Aniello, Roberto Baldoni

RICHIAMI DI RETE

Need for a Protocol Architecture

- The procedures involved in exchanging data between devices can be complex
- There must be a data path between the two computers, either directly or via a communication network
- Typical tasks performed are:
    - source system must either activate the direct data communication path or inform the communication network of the identity of the desired destination system
    - source system must ascertain that the destination system is prepared to receive data
    - file transfer application on the source system must ascertain that the file management program on the destination system is prepared to accept and store the file for this particular user
    - if the file formats or data representations used on the two systems are incompatible, one or the other system must perform a format translation function

Computer Communications

- The exchange of information between computers for the purpose of cooperative action is
  computer communications
- When two or more computers are interconnected via a communication network, the set of computer stations is referred to as a
  computer network

- In discussing computer communications and computer networks, two concepts are paramount:
  1) protocols
  2) computer communications architecture
  (or protocol architecture)

Protocol

- A set of rules governing the exchange of data between two entities
    - used for communication between entities in different systems
  - Entity: anything capable of sending or receiving information
  - System: physically distinct object that contains one or more entities
  - To communicate successfully, two entities must “speak the same language”

- Key elements of a protocol are:

Syntax	Semantics	Timing
• includes such things as data format and signal levels	• includes control information for coordination and error handling	• includes speed matching and sequencing

Protocol Architecture

There must be a high degree of cooperation between the two computer systems

→ Rather than implementing everything as a single module, tasks are broken up into subtasks

Example

The file transfer module contains all the logic that is unique to the file transfer application

↓

The communications service module has to assure that the two computer systems are active and ready for data transfer, and for keeping track of the data that are being exchanged to assure delivery

↓

The logic for actually dealing with the network is put into a separate network access module

The structured set of modules that implements the communications function is referred to as a ***protocol architecture***

File Transfer Architecture

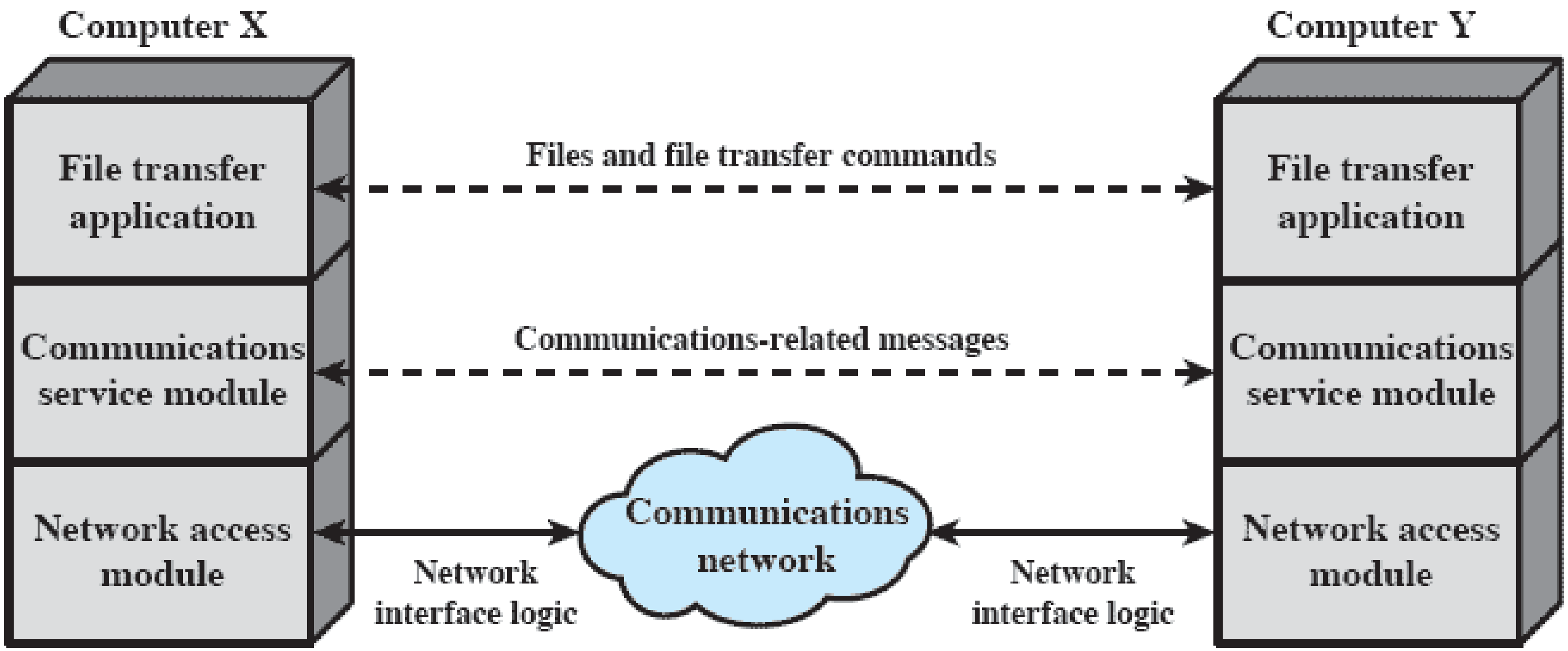


Figure 17.1 A Simplified Architecture for File Transfer

TCP/IP
Protocol Architecture

- A result of protocol research and development conducted on the experimental packet-switched network, ARPANET
- Referred to as the TCP/IP protocol suite
- The communication task for TCP/IP can be organized into five independent layers:
  - application layer
  - host-to-host, or transport layer
  - internet layer
  - network access layer
  - physical layer

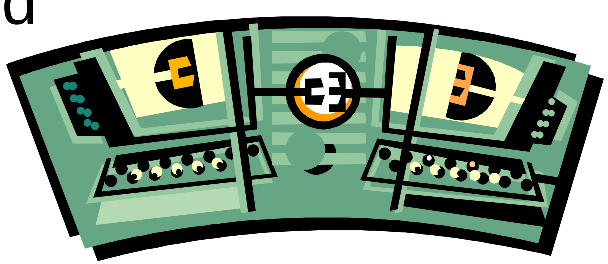
Physical Layer and Network Access Layer

Physical layer

- covers the physical interface between a data transmission device and a transmission medium or network
- this layer is concerned with:
  - specifying the characteristics of the transmission medium
  - the nature of the signals
  - the data rate

Network access layer

- concerned with the exchange of data between an end system and the network to which it is attached
- the specific software used at this layer depends on the type of network to be used



Internet Layer and Transport Layer

Internet layer

- function is to allow data to traverse multiple interconnected networks
- the ***Internet Protocol (IP)*** is used at this layer to provide the routing function across multiple networks
  - implemented not only in the end systems but also in routers
    - » a ***router*** is a processor that connects two networks
    - » primary function is to relay data from one network to the other

Transport layer

- a common layer shared by all applications and contains the mechanisms for providing reliability when data is exchanged
- the ***transmission control protocol (TCP)*** is the most commonly used protocol to provide this function

Application Layer

- contains the logic needed to support the various user applications
- for each different type of application, a separate module is needed that is peculiar to that application

TCP/IP Concepts

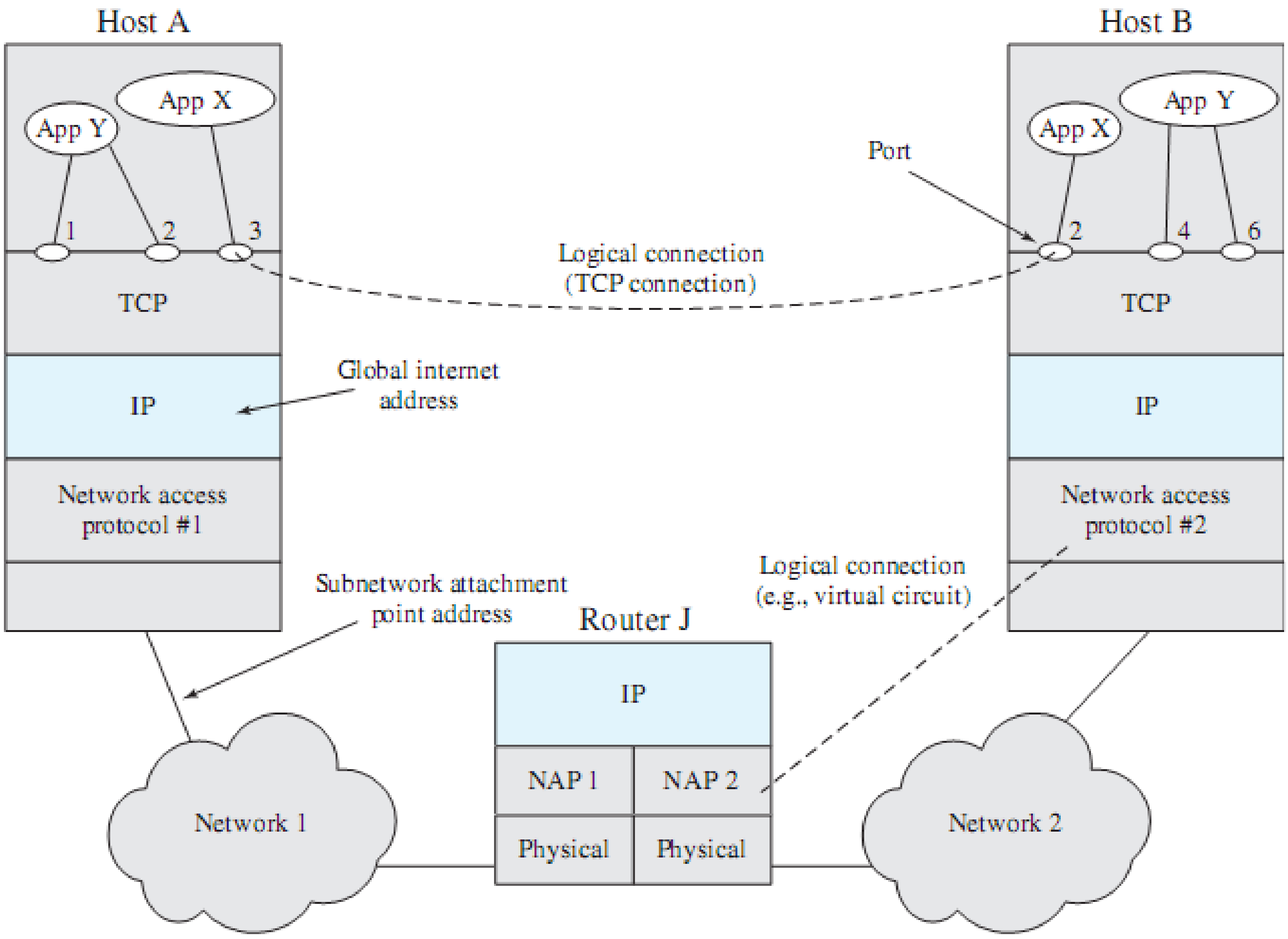


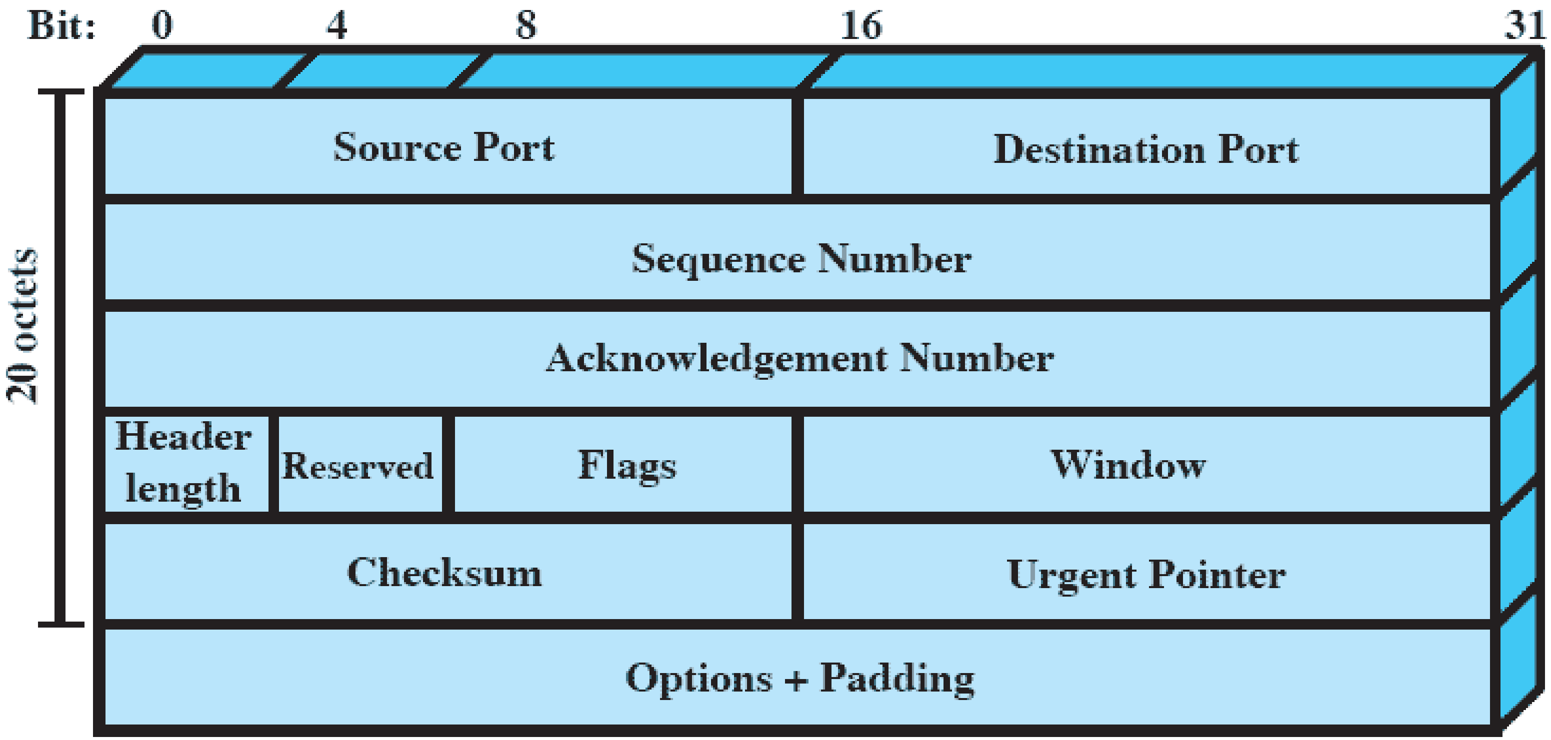
Figure 17.4 TCP/IP Concepts

Operation of TCP/IP

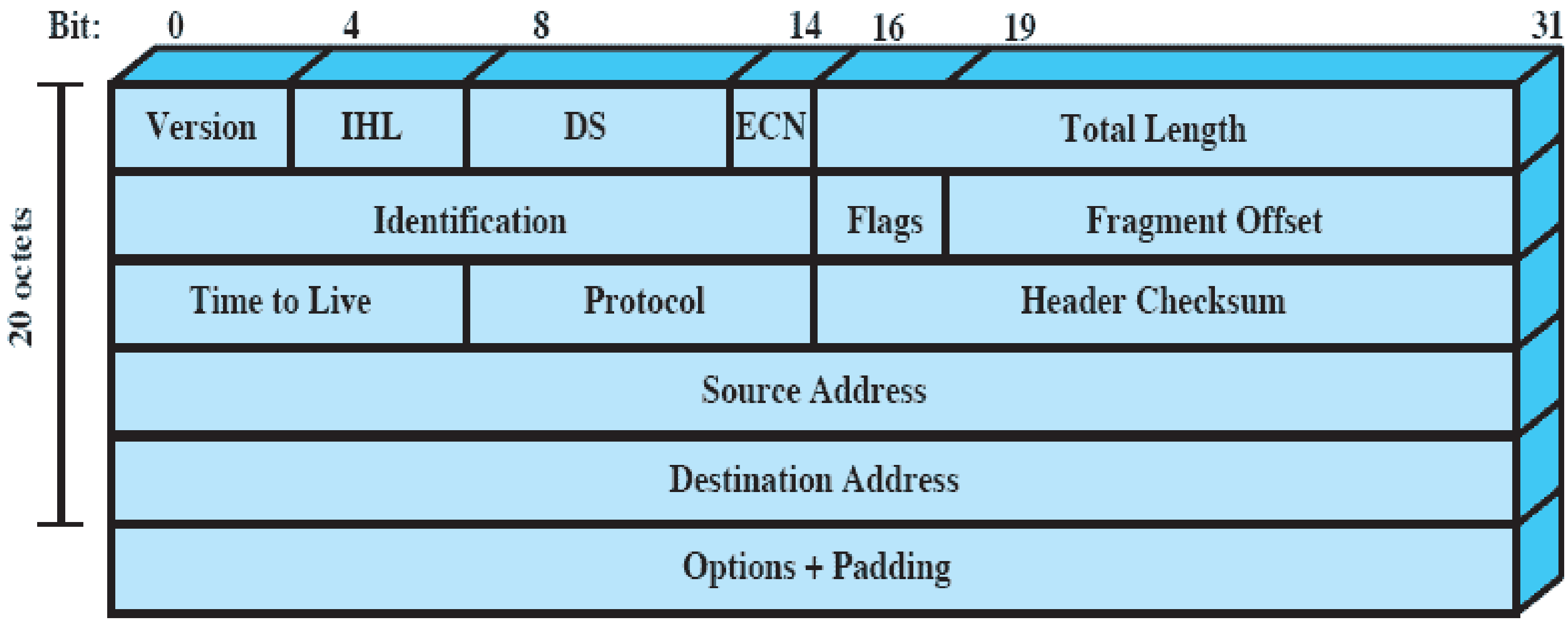
- Every entity in the overall system must have a unique address
  - two levels of addressing are needed

1. Each host on a network must have a unique global internet address
   - » this address is used by IP for routing and delivery
2. Each application within a host must have an address that is unique within the host
   - » this allows the host-to-host protocol (TCP) to deliver data to the proper process
   - » these addresses are known as ***ports***

TCP header


Bit: 0 4 8 16 31
20 octets
Source Port
Destination Port
Sequence Number
Acknowledgement Number
Header length
Reserved
Flags
Window
Checksum
Urgent Pointer
Options + Padding

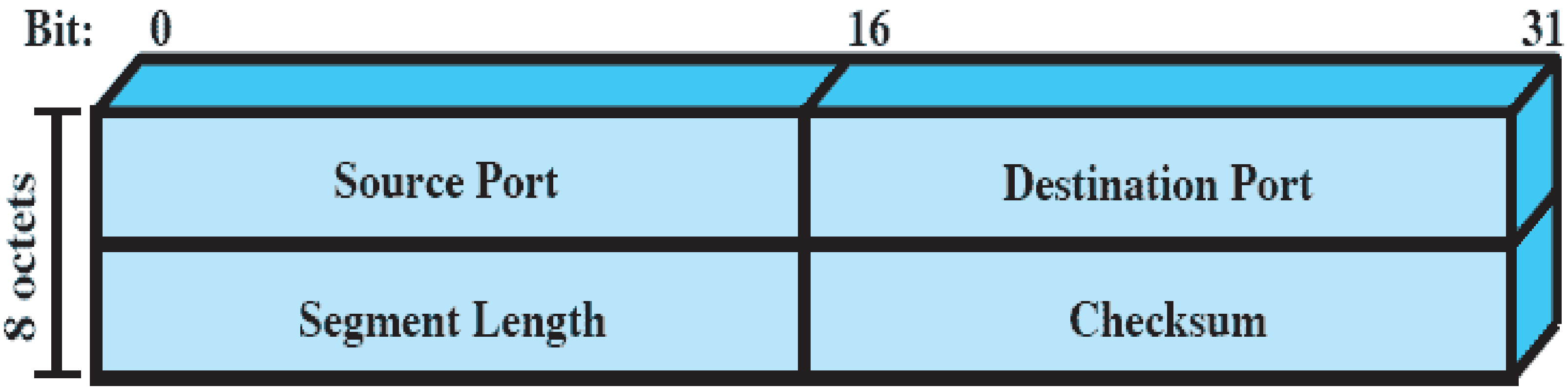
IP header (IPv4)



TCP/IP Applications

- A number of applications have been standardized to operate on top of TCP
- Examples:
  - **Simple Mail Transfer Protocol (SMTP)**
    - provides a basic electronic mail facility
    - features include mailing lists, return receipts, and forwarding
  - **File Transfer Protocol (FTP)**
    - used to send files from one system to another under user command
    - both text and binary files are accommodated
    - provides features for controlling user access
  - **Secure Shell (SSH)**
    - provides a secure remote logon capability, which enables a user at a terminal or personal computer to log on to a remote computer and function as if directly connected to that computer
    - supports file transfer between the local host and a remote server
    - SSH traffic is carried on a TCP connection

UDP



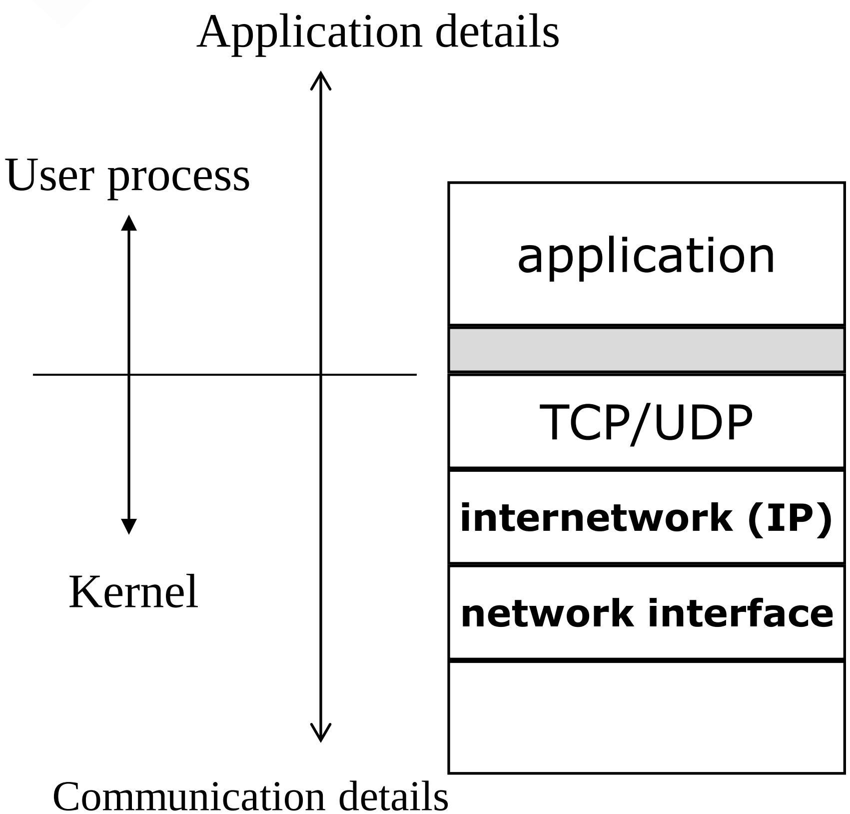
- minimum protocol mechanism
  - **connectionless**
  - no guarantees about delivery, preservation of sequence, nor protection against duplication
  - useful, e.g., for transaction-oriented applications
  - multicast support

SOCKETS

Sockets

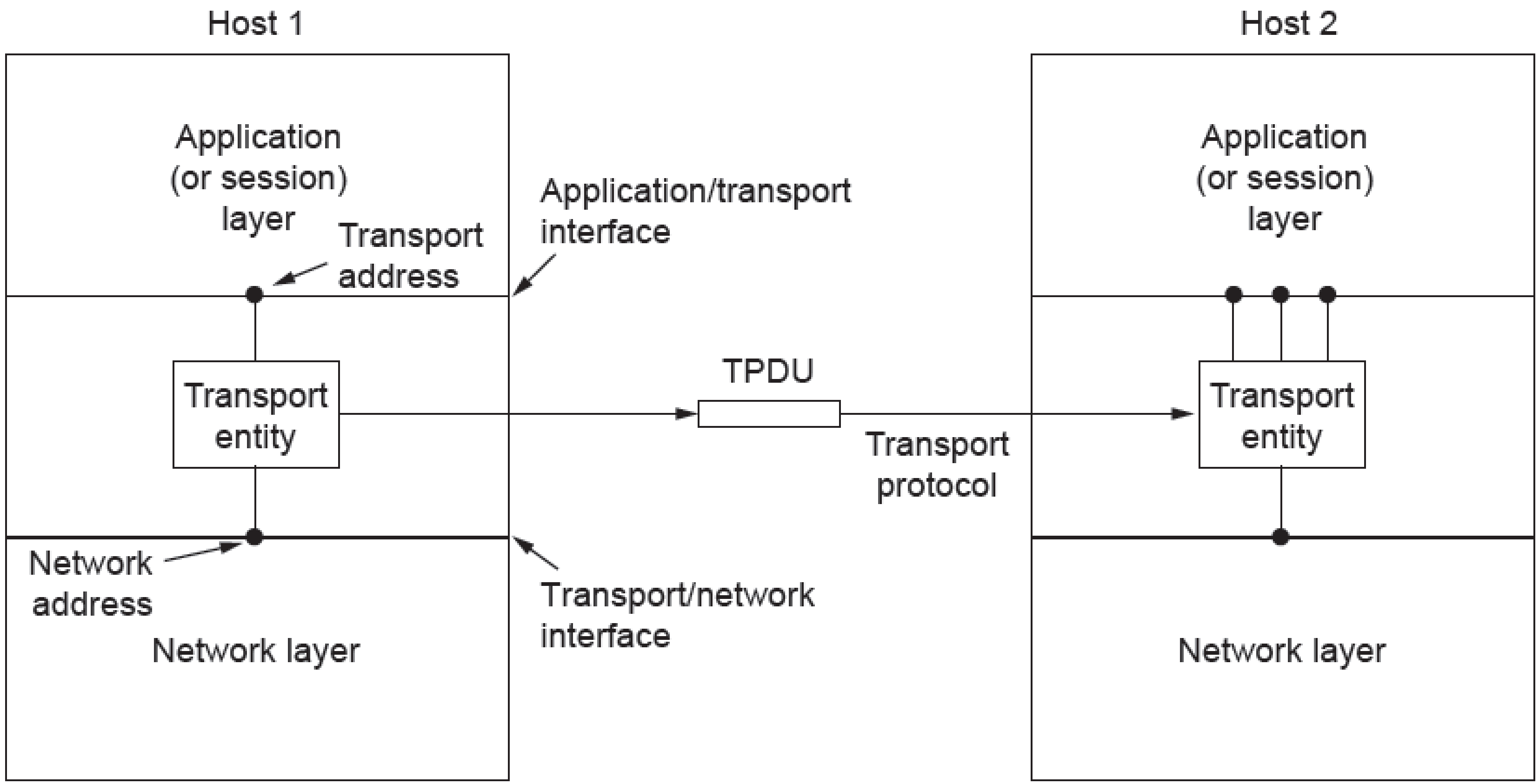
- Concept was developed in the 1980s in the UNIX environment as the Berkeley Sockets Interface (BSI)
- Enables communication between a client and server process
- May be either connection oriented or connectionless
- Can be considered an endpoint in communication
- The BSI transport layer interface is the de-facto standard API for developing networking applications
- Windows sockets (WinSock) are based on the Berkeley specification

Sockets



Interfaccia socket
TLI (Transport Layer Interface)

Services Provided to the Upper Layers



The network, transport, and application layers

The Socket

- ***IP addresses*** identify the respective host systems
- The concatenation of a port value and an IP address forms a ***socket***, which is unique throughout the Internet

Stream sockets

- makes use of TCP
- provides a connection-oriented reliable data transfer

Datagram sockets

- make use of UDP
- delivery is not guaranteed, nor is order necessarily preserved

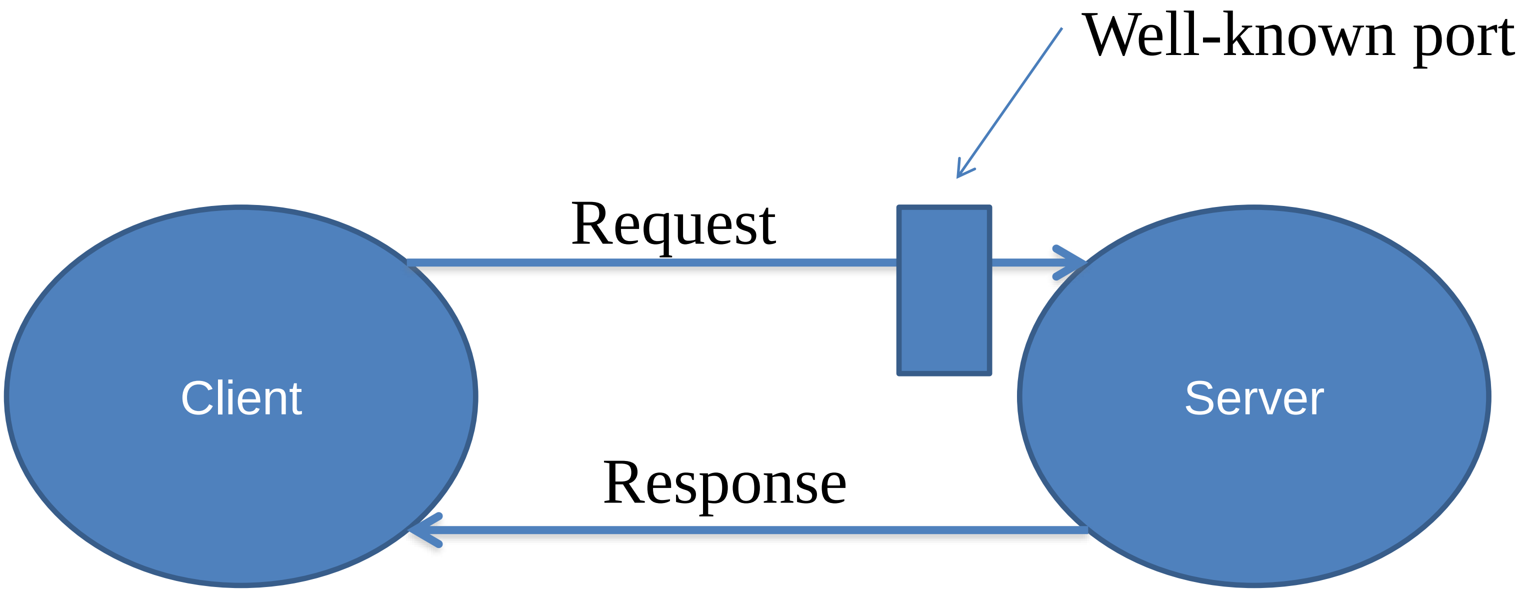
Raw sockets

- allow direct access to lower-layer protocols

Socket Basics

- An end-point for a IP network connection
  - what the application layer “plugs into”
  - programmer cares about API
- End point determined by two things:
  - Host address: IP address is *Network Layer*
  - Port number: is *Transport Layer*
- Two end-points determine a connection: socket pair
  - ex: 206.62.226.35,p21 + 198.69.10.2,p1500
  - ex: 206.62.226.35,p21 + 198.69.10.2,p1499

Client-Server Approach



Ports

BSD res. Ports	BSD ephemeral ports	BSD servers (non-privileged)
0 ↔ 1023	1024 ↔ 5000	5001 ↔ 65535

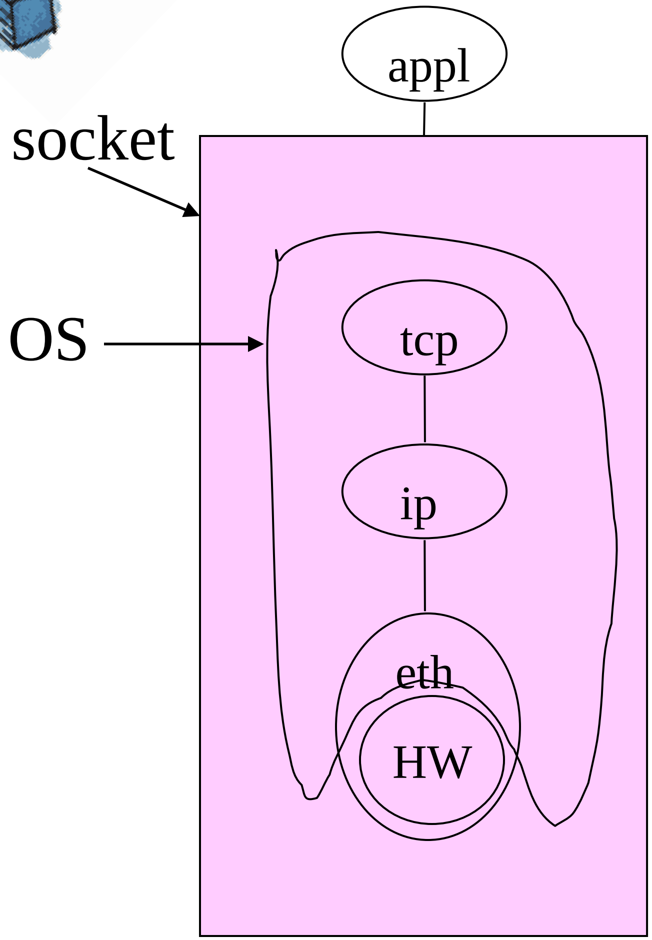
Ports

- Numbers (vary in BSD, Solaris, Linux):
  - 0-1023 “reserved”, must be root
  - 1024 - 5000 “ephemeral” (short-lived ports assigned automatically by the OS to clients)
  - however, many systems allow > 3977 ephemeral ports due to the number of increasing handled by a single PC. (Sun Solaris provides 30,000 in the last portion of BSD non-privileged)

- Well-known, reserved services /etc/services:
  - ftp 21/tcp
  - telnet 23/tcp
  - finger 79/tcp
  - snmp 161/udp

- Several client program needs to be a server at the same time (rlogin, rsh) as a part of the client-server authentication. These clients call the library function rresvport to create a socket and to assign an unused port in the range 512-1024.

Sockets and the OS



- User sees "descriptor", integer index
  - like: `FILE *`, or file index from `open()`
  - returned by `socket()` call (more later)

Transport Service Primitives

Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

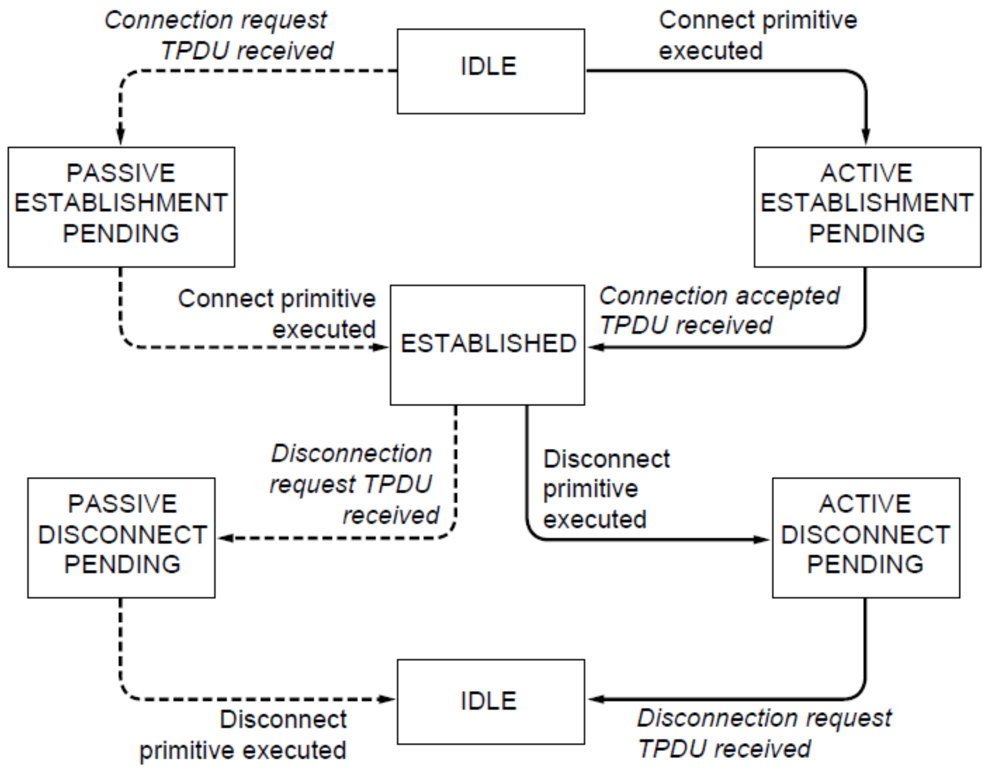
Primitives for a simple transport service

Socket Address Structure

```
struct in_addr {
    in_addr_t s_addr;              /* 32-bit IPv4 addresses */
};
struct sockaddr_in {
    uint8_t      sin_len;          /* length of structure */
    sa_family_t  sin_family;       /* AF_INET */
    in_port_t    sin_port;         /* TCP/UDP Port num */
    struct in_addr sin_addr;       /* IPv4 address (above) */
    char sin_zero[8];              /* unused */
}
```

- **Zero-initialize (e.g., bzero, memset) before using sockaddr**

Berkeley Sockets (1)



A state diagram for a simple connection management scheme. Transitions labeled in italic are caused by packet arrivals. The solid lines show the client's state sequence. The dashed lines show the server's state sequence.

Berkeley Sockets (2)

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

The socket primitives for TCP

Socket Connection

- For a stream socket, once the socket is created, a connection must be set up to a remote socket
- One side functions as a client and requests a connection to the other side, which acts as a server

Client:

issues a connect() that specifies both a local socket and the address of a remote socket

once a connection is set up, getpeername() can be used to find out who is on the other end of the connected stream socket

Server side of a connection setup requires two steps:

a server application issues a listen (), indicating that the given socket is ready to accept incoming connections

each incoming connection is placed in this queue until a matching accept () is issued by the server side

Addresses and Sockets

- Structure to hold address information
- Functions pass address from user to OS

  `bind()`

  `connect()` *(TCP only)*

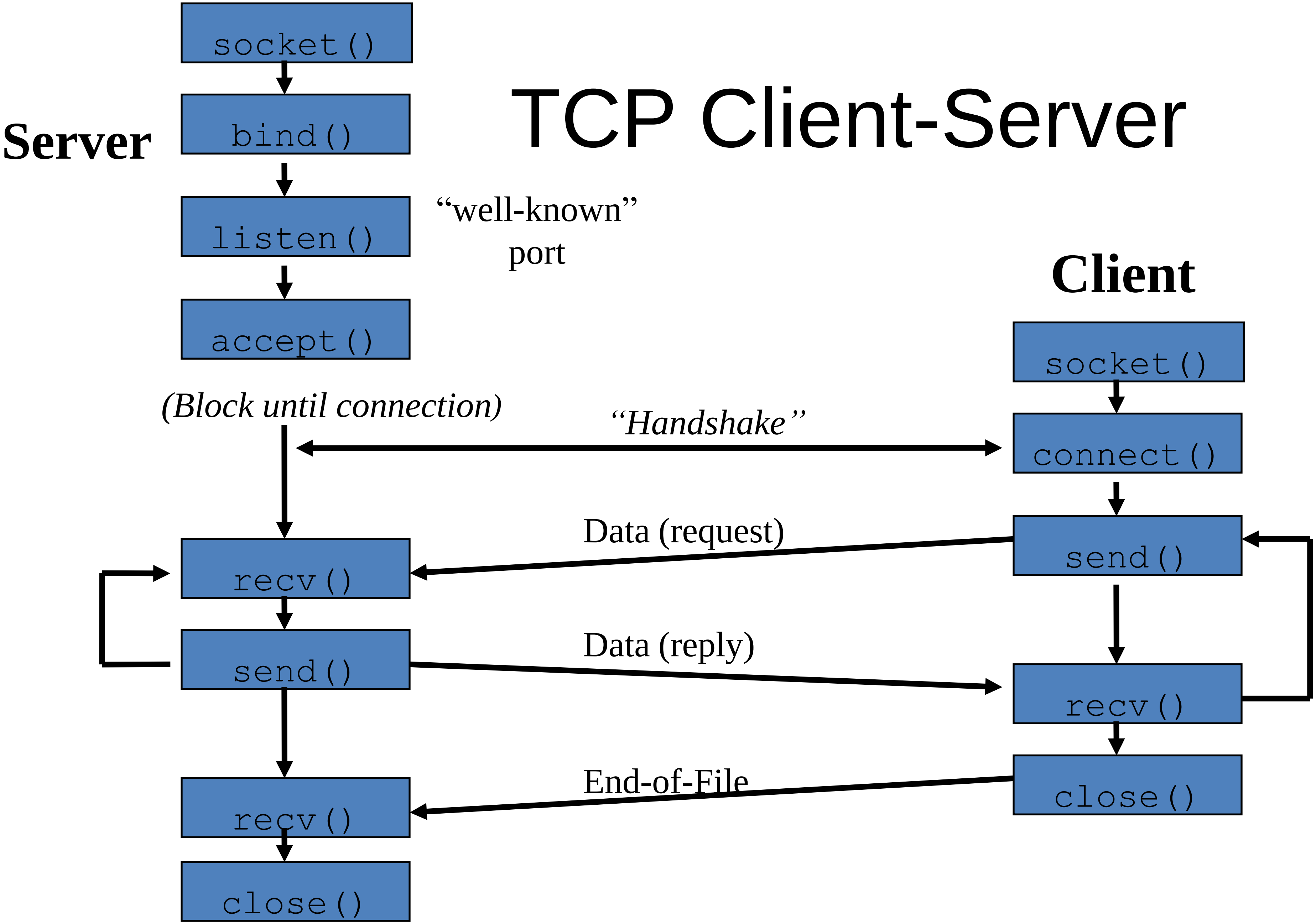
  `sendto()` *(UDP only)*

- Functions pass address from OS to user

  `accept()` *(TCP only)*

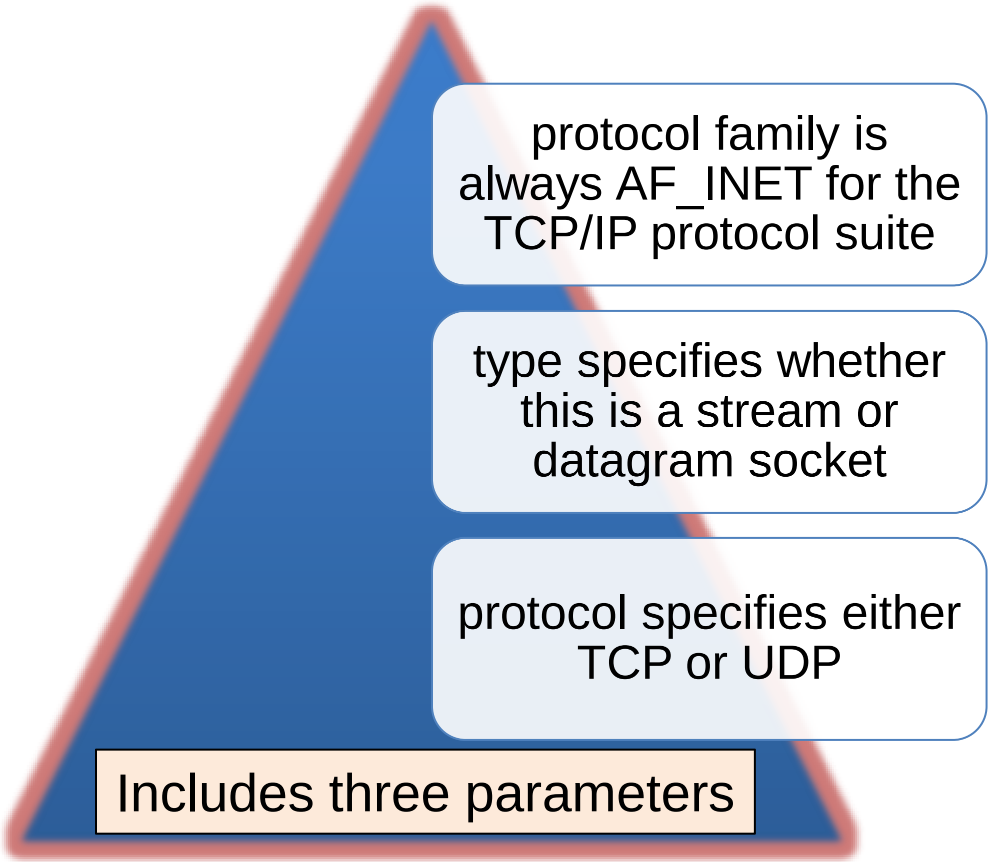
  `recvfrom()` *(UDP only)*

TCP Client-Server



Socket Setup: Socket()

- The first step in using Sockets is to create a new socket using the socket() API
- Returns an integer that identifies the socket
  - similar to a UNIX file descriptor



socket()

`int socket(int family, int type, int protocol);`

Create a socket, giving access to transport layer service

- *family* is one of
  - AF_INET (IPv4), AF_INET6 (IPv6), AF_LOCAL (local Unix),
  - AF_ROUTE (access to routing tables), AF_KEY (new, for encryption)
- *type* is one of
  - SOCK_STREAM (TCP), SOCK_DGRAM (UDP)
  - SOCK_RAW (for special IP packets, PING, etc. Must be root)
- *protocol* is 0 (used for some raw socket options)
- upon success returns socket descriptor
  - Integer, like file descriptor
  - Return -1 if failure

bind()

```
int bind(int sockfd, const struct sockaddr *myaddr,
        socklen_t addrlen);
```

Assign a local protocol address (“name”) to a socket.

- *sockfd* is socket descriptor from `socket()`
- *myaddr* is a pointer to address struct with:
  - *port number* and *IP address*
  - if port is 0, then host OS will pick ephemeral port
- *addrlen* is length of structure
- returns 0 if ok, -1 on error
  - EADDRINUSE (“Address already in use”)

Argomenti Valore-risultato

- Nelle chiamate che passano la struttura indirizzo da processo utente a OS (esempio bind) viene passato un puntatore alla struttura ed un intero che rappresenta il **sizeof** della struttura (oltre ovviamente ad altri parametri). In questo modo l'OS kernel sa esattamente quanti byte deve copiare nella sua memoria.
- Nelle chiamate che passano la struttura indirizzo dall'OS kernel al processo utente (esempio accept) vengono passati nella system call eseguita dall'utente un puntatore alla struttura ed un puntatore ad un intero in cui è stata preinserita la **dimensione della struttura** indirizzo. In questo caso, sulla chiamata della system call, il kernel dell'OS sa la dimensione della struttura quando la va a riempire e non ne oltrepassa i limiti. Quando la system call ritorna inserisce nell'intero la dimensione di quanti byte ha scritto nella struttura.
- Questo modo di procedere è utile in system call come la select e la getsockopt che vedrete durante le esercitazioni.

`listen()`

`int listen(int sockfd, int backlog);`

Change socket state for TCP server.

- *sockfd* is socket descriptor from `socket()`
- *backlog* is maximum number of *incomplete* connections
  - historically 5
  - implementation might use it just as a hint

accept()

```
int accept(int sockfd,
           struct sockaddr *cliaddr,
           socklen_t *addrlen);
```

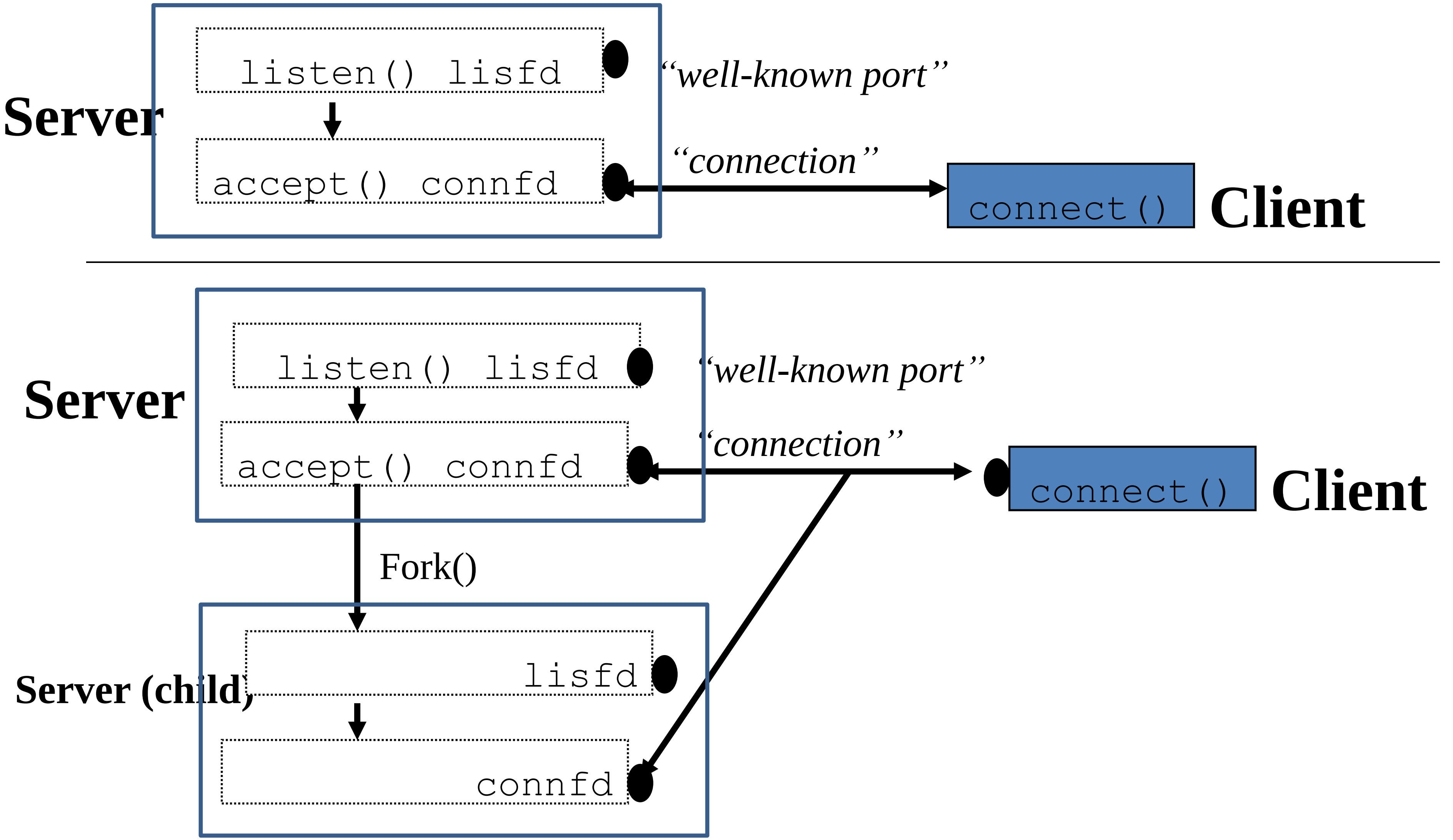
Return next completed connection.

- *sockfd* is socket descriptor from `socket()`
- *cliaddr* and *addrlen* return protocol address from client
- returns new descriptor, created by OS
- On error, -1 is returned, and *errno* is set appropriately.
- if used with `fork()`, can create concurrent server. If used with `pthread_create()` can create a multithread server

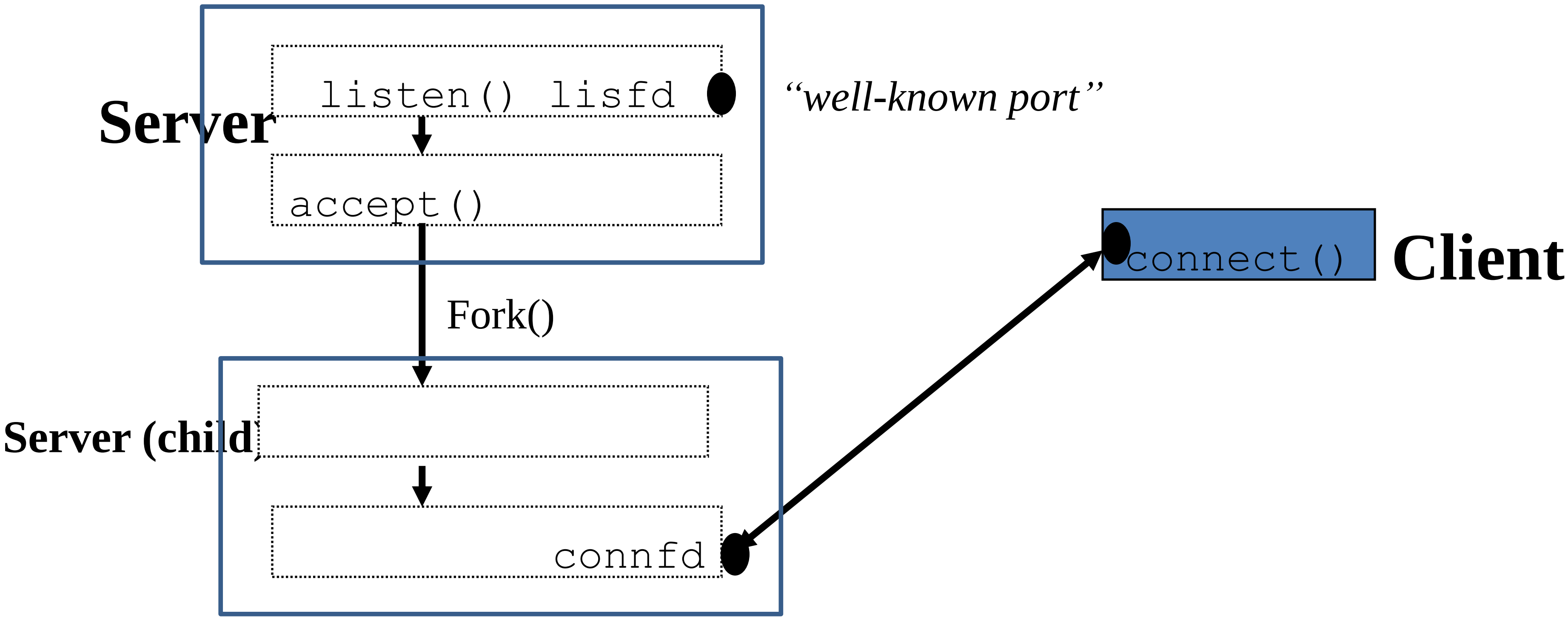
Accept()+fork()

```
lisfd = socket(....);
bind(lisfd,...);
listen(lisfd, 5);
while(1) {
   connfd = accept(lisfd,.....);
   if (pid = fork() == 0) {
      close(lisfd);
      doit(connfd);
      close(connfd);
      _exit(0);
   }
   close(connfd);
}
```

Status Client-Server after Fork returns



Status Client-Server after closing sockets



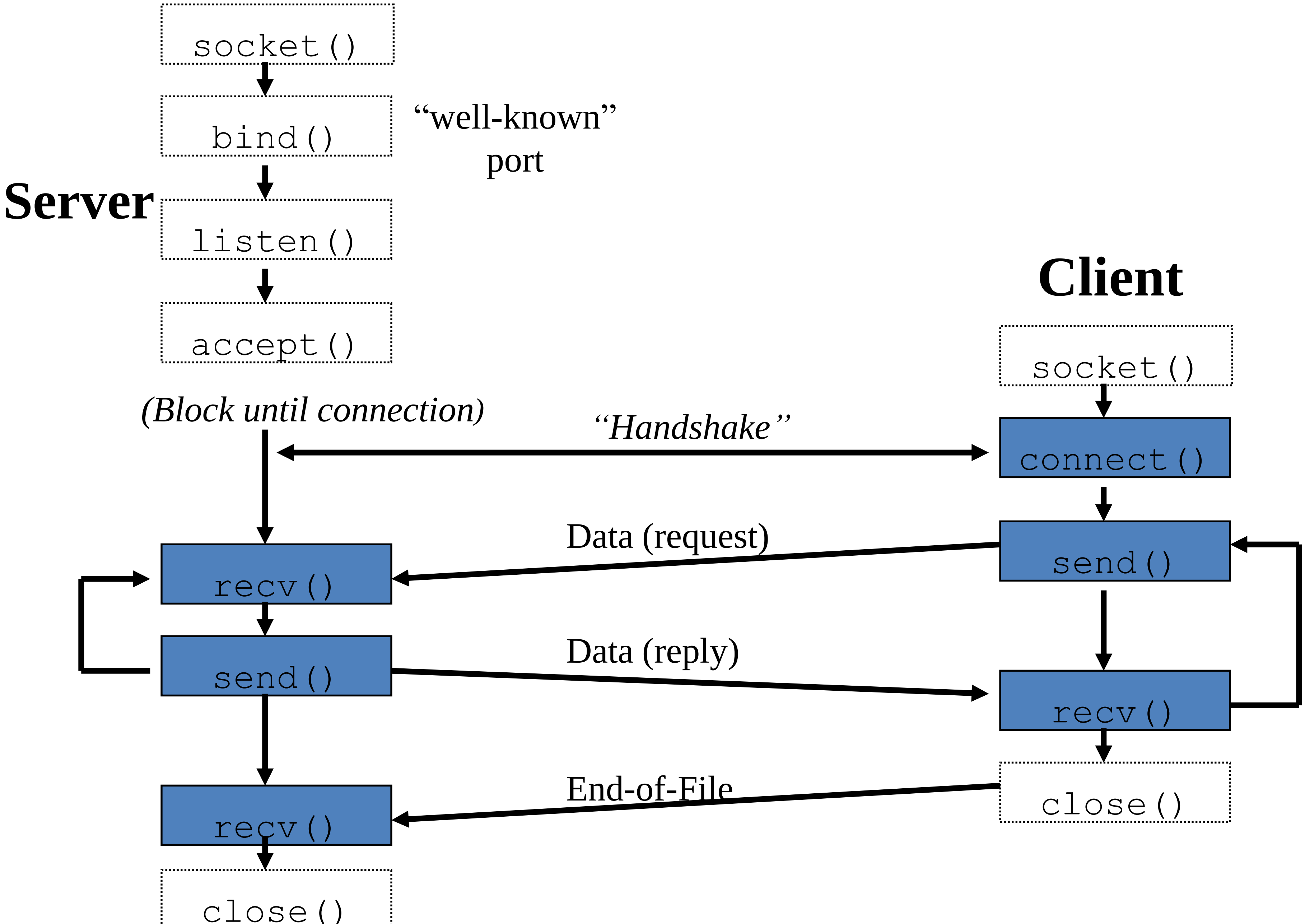
`close()`

`int close(int sockfd);`

Close socket for use.

- *sockfd* is socket descriptor from `socket()`
- closes socket for reading/writing
  – returns (doesn't block)
  – attempts to send any unsent data
  – socket option: SO_LINGER timeout
    - block until data sent
    - or discard any remaining data
  – Returns -1 if error

TCP Client-Server



`connect()`

```
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

Connect to server.

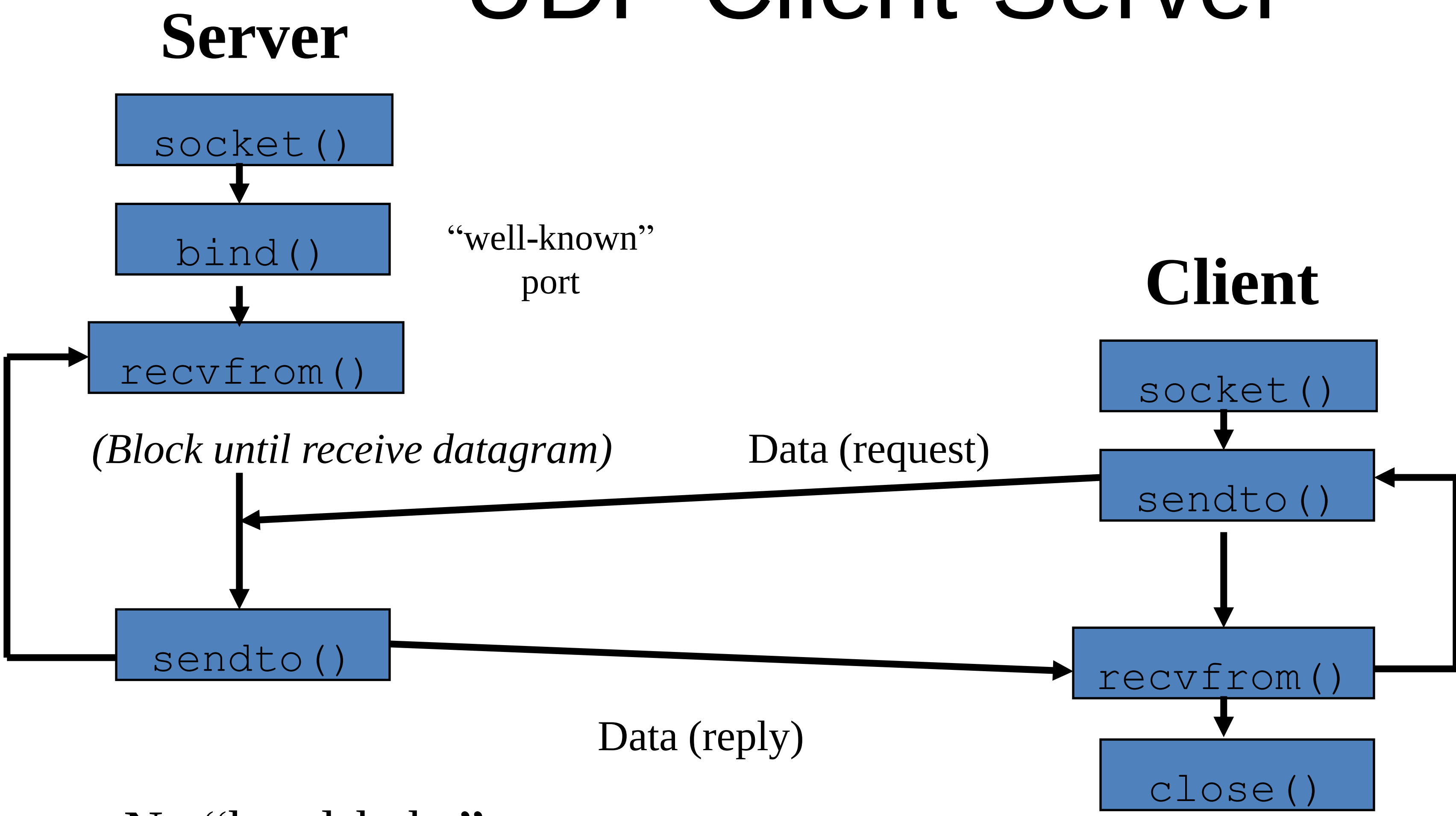
- *sockfd* is socket descriptor from `socket()`
- *servaddr* is a pointer to a structure with:
  - *port number* and *IP address*
  - must be specified (unlike `bind()`)
- *addrlen* is length of structure
- client doesn't need `bind()`
  - OS will pick ephemeral port
- returns socket descriptor if ok, -1 on error

Sending and Receiving

```
int recv(int sockfd, void *buff, size_t
  numBytes, int flags);

int send(int sockfd, void *buff, size_t
  numBytes, int flags);
```

- Same as `read()` and `write()` but for *flags*
  - MSG_DONTWAIT (this send non-blocking)
  - MSG_OOB (out of band data, 1 byte sent ahead)
  - MSG_PEEK (look, but don’t remove)
  - MSG_WAITALL (don’t give me less than max)
  - MSG_DONTROUTE (bypass routing table)

UDP Client-Server



- No "handshake"
- No simultaneous close
- No `fork`() for concurrent servers!

Sending and Receiving

```
int recvfrom(int sockfd, void *buff, size_t numBytes, int
   flags, struct sockaddr *from, socklen_t *addrlen);
```

```
int sendto(int sockfd, void *buff, size_t numBytes, int
   flags, const struct sockaddr *to, socklen_t addrlen);
```

- Same as `recv()` and `send()` but for *addr*
  - `recvfrom` fills in address of where packet came from
  - `sento` requires address of where sending packet to

gethostname()

- Get the name of the host the program is running on.
  - int gethostname(char *hostname, int bufferLength)
    - Upon return, hostname holds the name of the host
    - bufferLength provides a limit on the number of bytes that gethostname() can write to hostname.

Internet Address Library Routines (inet_addr() and inet_ntoa())

- unsigned long inet_addr(char *address);
  - converts address in dotted form to a 32-bit numeric value in network byte order
    - (e.g., “128.173.41.41”)
- char* inet_ntoa(struct in_addr address)
  - struct in_addr
    - address.s_addr is the long int representation

Domain Name Library Routine

- gethostbyname(): Given a host name (such as acavax.lynchburg.edu) get host information.
  – struct hostent* getbyhostname(const char *hostname)
    - char* h_name; // official name of host
    - char** h_aliases; // alias list
    - short h_addrtype; // address family (e.g., AF_INET)
    - short h_length;   // length of address (4 for AF_INET)
    - char** h_addr_list; // list of addresses (null pointer terminated)

Internet Address Library Routines

- Get the name of the host given its address
  - struct hostent* gethostbyaddr(
    const void *addr, int len, int type)
    - *addr is a pointer to a struct of a type depending on the address type (es: struct in_addr)
    - len is 4 if type is AF_INET
    - type is the address family (e.g., AF_INET)

WinSock

- Derived from Berkeley Sockets (Unix)
  - includes many enhancements for programming in the windows environment
- Open interface for network programming under Microsoft Windows
  - API freely available
  - Multiple vendors supply WinSock
  - Source and binary compatibility
- Collection of function calls that provide network services

Differences Between Berkeley Sockets and WinSock

Berkeley	WinSock
bzero()	memset()
close()	closesocket()
read()	not required
write()	not required
ioctl()	ioctlsocket()

Additional Features of WinSock 1.1

- WinSock supports three different modes
  - Blocking mode
    - socket functions don’t return until their jobs are done
    - same as Berkeley sockets
  - Non-blocking mode
    - Calls such as accept() don’t block, but simply return a status
  - Asynchronous mode
    - Uses Windows messages
      - FD_ACCEPT - connection pending
      - FD_CONNECT - connection established
      - etc.

WinSock 2

- Supports protocol suites other than TCP/IP
  - DecNet
  - IPX/SPX
  - OSI
- Supports network-protocol independent applications
- Backward compatible with WinSock 1.1

WinSock 2 (Continued)

- Uses different files
  - winsock2.h
  - different DLL (WS2_-32.DLL)
- API changes
  - accept() becomes WSAAccept()
  - connect() becomes WSAConnect()
  - inet_addr() becomes WSAAddressToString()
  - etc.

Stream Sockets in Java

```java
public class GreetServer {
    private ServerSocket serverSocket;
    private Socket clientSocket;
    private PrintWriter out;
    private BufferedReader in;

    public void start(int port) {
        serverSocket = new ServerSocket(port);
        clientSocket = serverSocket.accept();
        out = new PrintWriter(clientSocket.getOutputStream(), true);
        in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        String greeting = in.readLine();
            if ("hello server".equals(greeting)) {
                out.println("hello client");
            }
            else {
                out.println("unrecognised greeting");
            }
    }

    public void stop() {
        in.close();
        out.close();
        clientSocket.close();
        serverSocket.close();
    }
    public static void main(String[] args) {
        GreetServer server=new GreetServer();
        server.start(6666);
        server.stop();
    }
}
```

Obviously a well done server must include

- Process/thread generation
- Continuos listening
- Repeated communication

Stream Sockets in Java

```java
public class GreetClient {
    private Socket clientSocket;
    private PrintWriter out;
    private BufferedReader in;

    public void startConnection(String ip, int port) {
        clientSocket = new Socket(ip, port);
        out = new PrintWriter(clientSocket.getOutputStream(), true);
        in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
    }

    public String sendMessage(String msg) {
        out.println(msg);
        String resp = in.readLine();
        return resp;
    }

    public void stopConnection() {
        in.close();
        out.close();
        clientSocket.close();
    }

    public static void main(String[] args) {
        GreetClient client = new GreetClient();
        client.startConnection("127.0.0.1", 6666);
        String response = client.sendMessage("hello server");
        assertEquals("hello client", response);
        client.stopConnection()
    }
}
```

UDP Sockets in Java

```java
class UDPServer {
 public static void main(String args[]) throws Exception
 {

    DatagramSocket serverSocket = new DatagramSocket(9876);
    byte[] receiveData = new byte[1024];
    byte[] sendData = new byte[1024];

    while(true){

      DatagramPacket receivePacket =
         new DatagramPacket(receiveData, receiveData.length);
      serverSocket.receive(receivePacket);

      String sentence = new String(receivePacket.getData());
      InetAddress IPAddress = receivePacket.getAddress();
      int port = receivePacket.getPort();

      String capitalizedSentence = sentence.toUpperCase();
      sendData = capitalizedSentence.getBytes();

      DatagramPacket sendPacket =
         new DatagramPacket(sendData, sendData.length, IPAddress, port);

      serverSocket.send(sendPacket);
    }
 }
}
```

Stream Sockets in Java

```
class UDPClient {
 public static void main(String args[]) throws Exception
 {

   BufferedReader inFromUser =
     new BufferedReader(new InputStreamReader(System.in));

   DatagramSocket clientSocket = new DatagramSocket();

   InetAddress IPAddress = InetAddress.getByName("hostname");

   byte[] sendData = new byte[1024];
   byte[] receiveData = new byte[1024];

   String sentence = inFromUser.readLine();
   sendData = sentence.getBytes();
   DatagramPacket sendPacket =
     new DatagramPacket(sendData, sendData.length, IPAddress, 9876);

   clientSocket.send(sendPacket);

   DatagramPacket receivePacket =
     new DatagramPacket(receiveData, receiveData.length);

   clientSocket.receive(receivePacket);

   String modifiedSentence = new String(receivePacket.getData());

   System.out.println("FROM SERVER:" + modifiedSentence);
   clientSocket.close();
 }
}
```

Stream Sockets in Python

```
#!/usr/bin/env python3
# server

import socket

HOST = '127.0.0.1'  # Standard loopback interface address (localhost)
PORT = 65432        # Port to listen on (non-privileged ports are > 1023)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen()
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data:
            break
        conn.sendall(data)
s.close()
```

Obviously a well done server must include

- Process/thread generation
- Continuos listening
- Repeated communication

Stream Sockets in Python

```
#!/usr/bin/env python3
# client

import socket

HOST = '127.0.0.1'  # The server's hostname or IP address
PORT = 65432        # The port used by the server

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.sendall(b'Hello, world')
data = s.recv(1024)

print('Received', repr(data))
s.close()
```

UDP Sockets in Python

```
#!/usr/bin/env python3
# server

import socket

UDP_IP = "127.0.0.1"
UDP_PORT = 5005

sock = socket.socket(socket.AF_INET, # Internet
                     socket.SOCK_DGRAM) # UDP
sock.bind((UDP_IP, UDP_PORT))

while True:
    data, addr = sock.recvfrom(1024) # buffer size is 1024 bytes
    print "received message:", data
    sock.sendto(data, addr)
```

Obviously a well done server must include

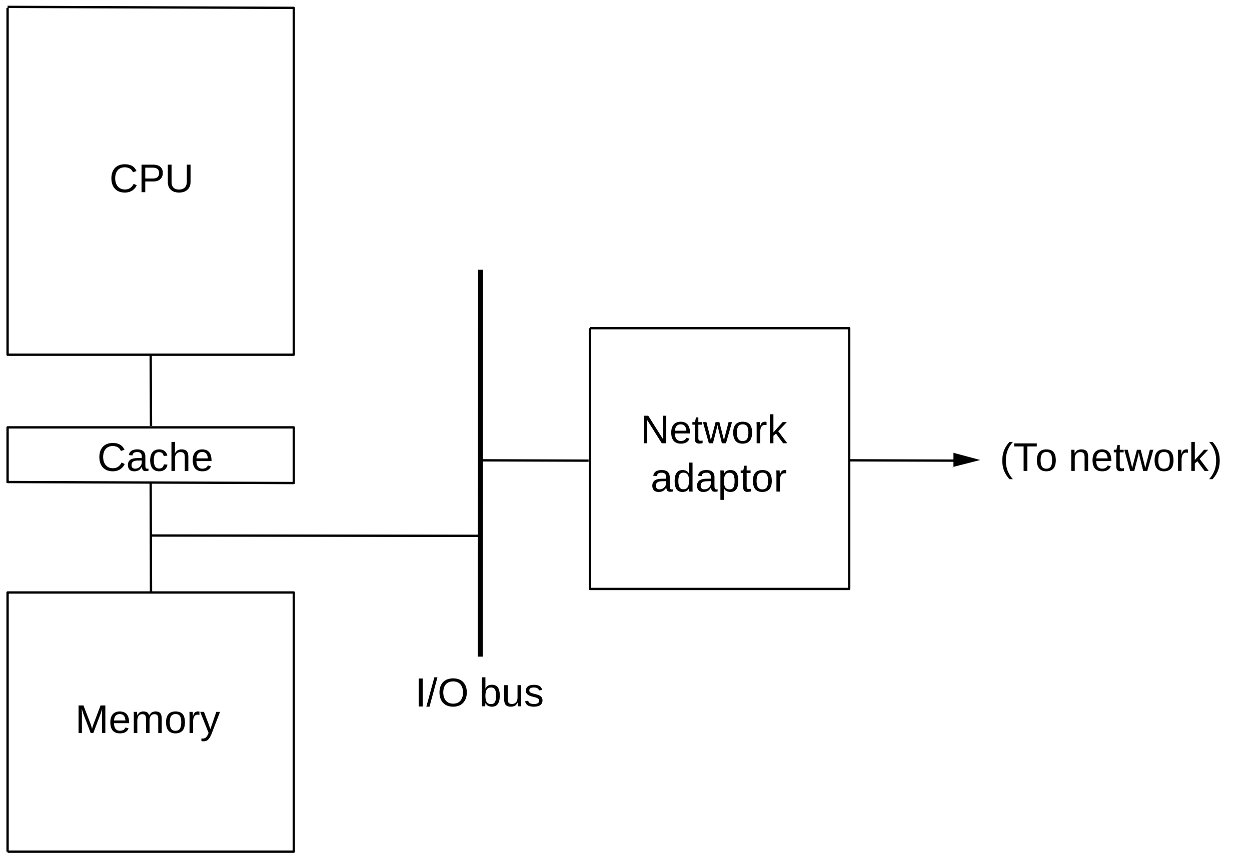
- Process/thread generation
- Continuos listening
- Repeated communication

UDP Sockets in Python

```
#!/usr/bin/env python3
# client

import socket

UDP_IP = "127.0.0.1"
UDP_PORT = 5005
MESSAGE = "Hello, World!"

print "UDP target IP:", UDP_IP
print "UDP target port:", UDP_PORT
print "message:", MESSAGE

sock = socket.socket(socket.AF_INET, # Internet
                     socket.SOCK_DGRAM) # UDP
sock.sendto(MESSAGE, (UDP_IP, UDP_PORT))
data, addr = sock.recvfrom(1024)
sock.close()
```

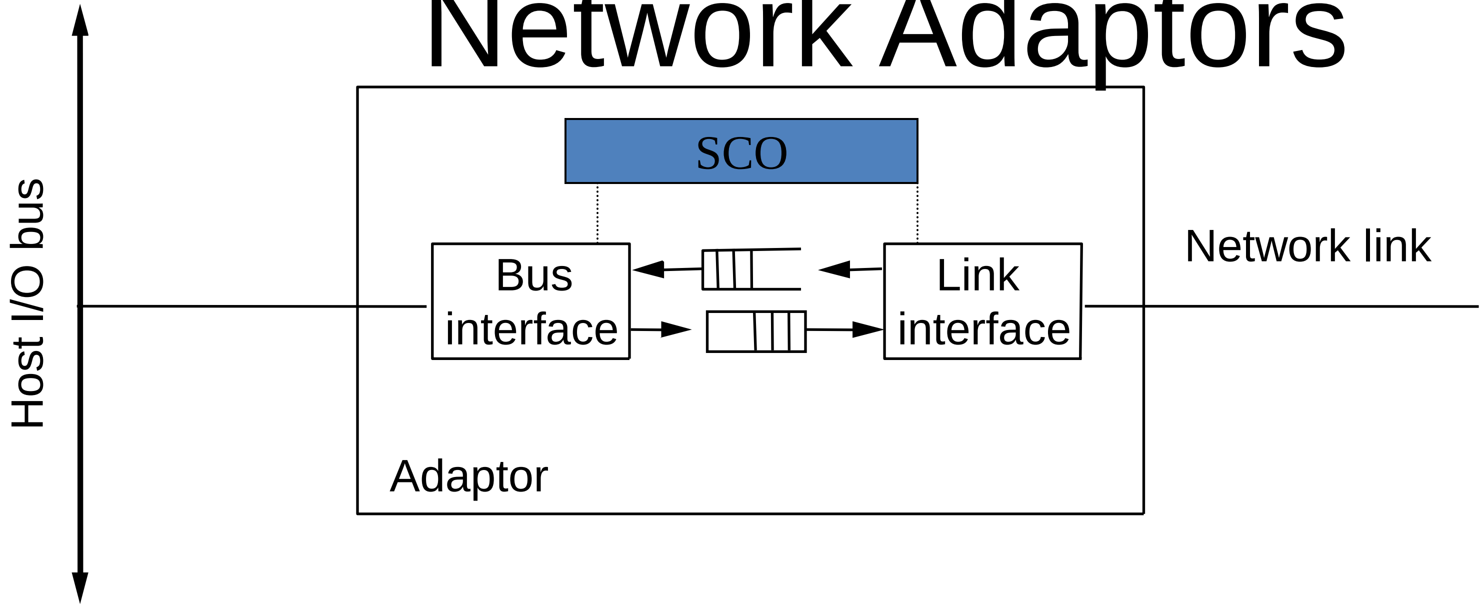
Network Adaptors

Network Adaptors



Interfaccia tra Host e Rete

Network Adaptors



La scheda è costituita da due parti separate che interagiscono attraveso una FIFO che maschera l’asincronia tra la rete e il bus interno

- La prima parte interagisce con la CPU della scheda
- La seconda parte interagisce con la rete (implementando il livello fisico e di collegamento)

Tutto il sistema è comandato da una SCO (sottosistema di controllo della scheda)

Vista dall’host

- L’adaptor esporta verso la CPU uno o più registri macchina (Control Status Register)
- CSR è il mezzo di comunicazione tra la SCO della scheda e la CPU

```
/* CSR
* leggenda: RO - read only; RC - Read/Clear (writing 1 clear, writing 0 has no effect);
* W1 write-1-only (writing 1 sets, writing 0 has no effect)
* RW - read/write; RW1 - Read-Write-1-only
*/
#define LE_ERR 0X8000
…………………………………………….
#define LE_RINT 0X0400  /* RC richiesta di interruzione per ricevere un pacchetto */
#define LE_TINT 0X0200  /* RC pacchetto trasmesso */
………………………………………………..
#define LE_INEA 0X0040  /* RW abilitazione all’emissione di un interrupt da parte
                           dell’adaptor verso la CPU */
………………………………….
#define LE_TDMD 0X0008  /* W1 richiesta di trasmissione di un pacchetto dal device
                           driver verso l’adaptor */

……………………………………………………………
```

Vista dall'host

L'host può controllare cosa accade in CSR in due modi

- Busy waiting (la CPU esegue un test continuo di CSR fino a che CSR non si modifica indicando la nuova operazione da eseguire. Ragionevole solo per calcolatori che non devono fare altro che attendere e trasmettere pacchetti, ad esempio i router)
- Interrupt (l'adaptor invia un interrupt all'host, il quale fa partire un *interrupt handler* che va a leggere CSR per capire l'operazione da fare)

Trasferimento dati da adaptor a memoria (e viceversa)

- Direct Memory Access
  - Nessun coinvolgimento della CPU nello scambio dati
  - Il SO assegna un’area di memoria all’host
  - Frame inviati immediatamente alla memoria di lavoro dell’host
  - Pochi bytes di memoria necessari sulla scheda

- Programmed I/O
  - Lo scambio dati tra memoria e adaptor passa per la CPU
  - Impone di bufferizzare almeno un frame sull’adaptor
  - La memoria deve essere di tipo dual port
    - Il processore e l’adaptor possono sia leggere che scrivere in questa porta

DMA: Buffer Descriptor list (BD)

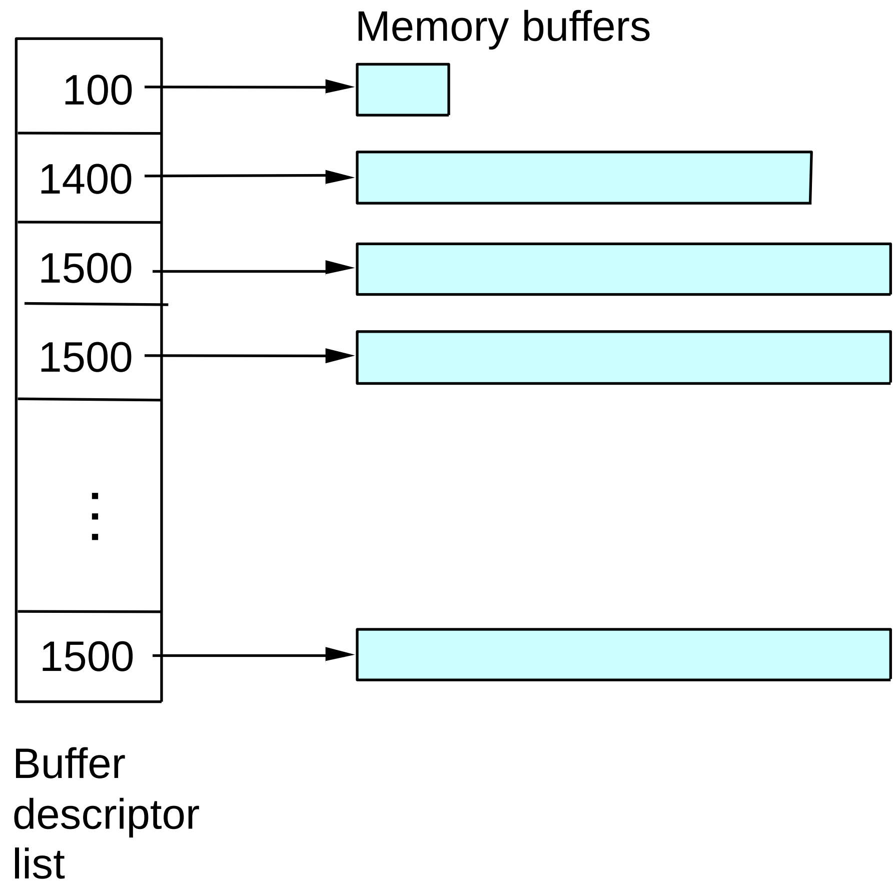
La memoria dove allocare i frames è organizzata attraverso una ***buffer descriptor list***

1 in scrittura

1 in lettura

Un vettore di puntatori ad aree di memoria (buffers) dove è descritta la quantità di memoria disponibile in quell'area

In ethernet vengono tipicamente preallocati 64 buffers da 1500 bytes



Buffer Descriptor list

Tecnica usata per frame che arrivano dall'adaptor:

scatter read / gather write

- frame distinti sono allocati in buffer distinti
- un frame può essere allocato su più buffer (se più grande del buffer)
    - In ethernet non necessario

Viaggio di un messaggio all’interno dell’SO

Quando un messaggio viene inviato da un utente in un certo socket

1. Il SO copia il messaggio dal buffer della memoria utente in una zona di BD
2. Tale messaggio viene processato da tutti i livelli protocollari (esempio TCP, IP, device driver) che provvedono ad inserire gli opportuni header e ad aggiornare gli opportuni puntatori presenti nel BD in modo da poter sempre ricostruire il messaggio
3. Quando il messaggio ha completato l’attraversamento del protocol stack, viene avvertita la SCO dell’adaptor dal device driver attraverso il set dei bit del CSR (LE_TDMD e LE_INEA). Il primo invita la SCO ad inviare il messaggio sulla linea. Il secondo abilita la SCO ad inviare una interruzione
4. La SCO dell’adaptor invia il messaggio sulla linea
5. Una volta terminata la trasmissione, la SCO notifica il termine alla CPU attraverso il set del bit (LE_TINT) del CSR e scatena una interruzione
6. Tale interruzione avvia un interrupt handler che prende atto della trasmissione, resetta gli opportuni bit (LE_TINT e LE_INEA) e libera le opportune risorse (operazione semsignal su xmit_queue)

Device Drivers

Il device driver è una collezione di routine (inizializzare l’adaptor, invio di un frame sul link etc.) di OS che serve per “ancorare” il SO all’hardware sottostante specifico dell’adaptor

Esempio routine di <u>richiesta</u> di invio di un messaggo sul link

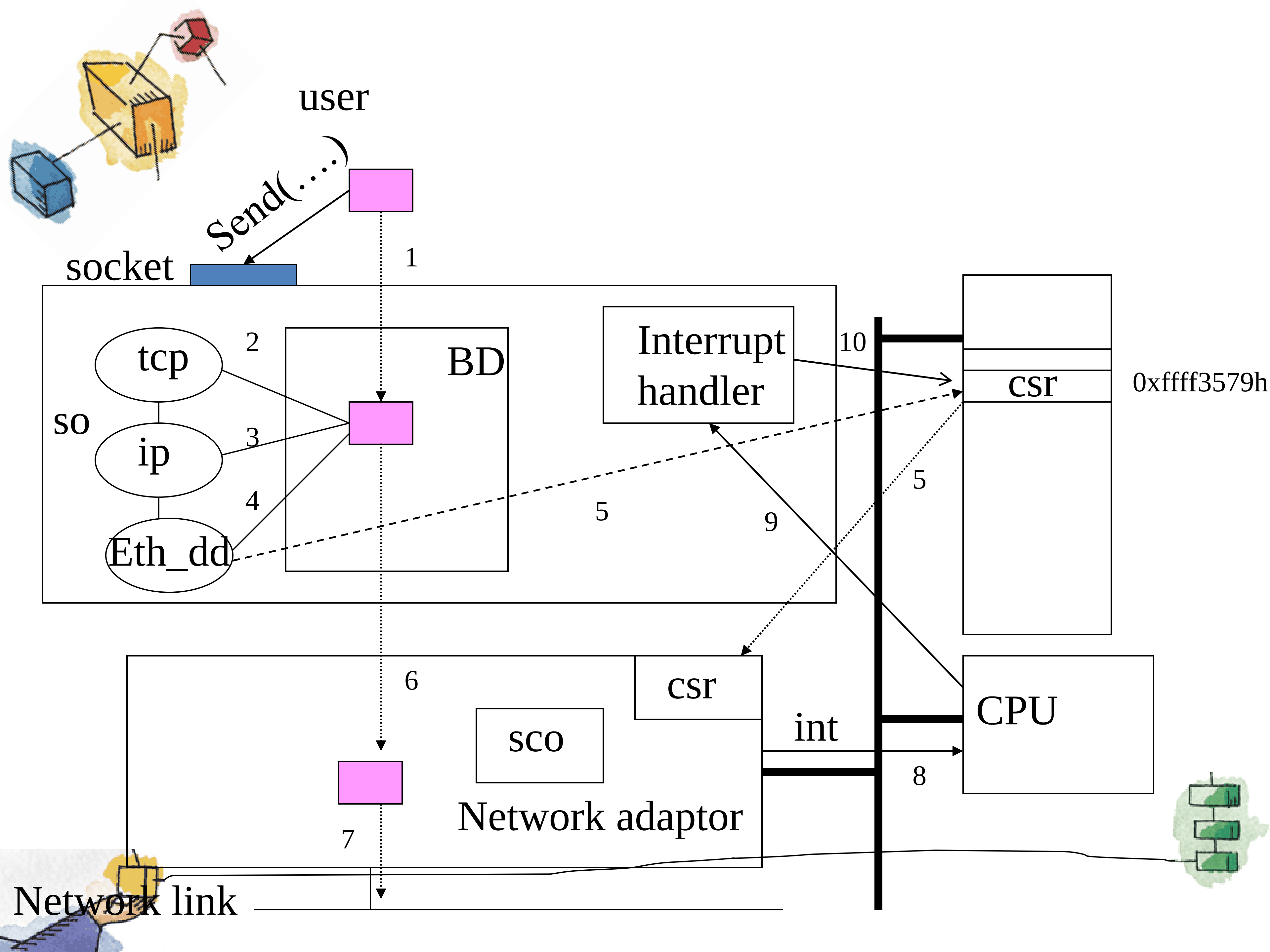
```
#define csr ((u_int) 0xffff3579 /*CSR address*/
Transmit(Msg *msg)
{    descriptor *d;
     semwait(xmit_queue);     /* abilita non piu’ di 64 accessi al BD*/
     d=next_desc();
     prepare_desc(d,msg);
     semwait(mutex);          /* abilità a non piu’ di un processo (dei potenziali 64)
                              alla volta la trasmissione verso l’adaptor */
     disable_interrupts();    /* il processo in trasmissione si protegge da eventuali
                              interruzioni dall’adaptor */
     csr= LE_TDMD | LE_INEA;  /* una volta preparato il messaggio invita la SCO dell’adaptor a
                              trasmetterlo e la abilita la SCO ad emettere una interruzione
                              una volta terminata la trasmissione */
     enable_interrupts();    /* riabilita le interruzioni */
     semsignal(mutex);        /* sblocca il semaforo per abilitare un altro processo a
}                             trasmettere */
```

“next_desc()” ritorna il prossimo buffer descriptor disponibile nel buffer descriptor list
“prepare_desc(d,msg)” il messaggio msg nel buffer d in un formato comprensibile dall’adaptor

Interrupt Handler

- Disabilita le interruzioni
- Legge il CSR per capire che cosa deve fare: tre possibilità
  1. C'è stato un errore
  2. Una trasmissione è stata completata
  3. Un frame è stato ricevuto
- Noi siamo nel caso 2
  - LE_TINT viene messo a zero (RC bit)
  - Ammette un nuovo processo nella BD poiché un frame è stato trasmesso
  - Abilita le interruzioni

```
lance_interrupt_handler()
{
    disable_interrupts();

    /* some error occurred */
    if (csr & LE_ERR)
    {
        print_error(csr);
        /* clear error bits */
        csr = LE_BABL | LE_CERR | LE_MISS | LE_MERR | LE_INEA;
        enable_interrupts();
        return();
    }

    /* transmit interrupt */
    if (csr & LE_TINT)
    {
        /* clear interrupt */
        csr = LE_TINT | LE_INEA;

        /* signal blocked senders */
        semSignal(xmit_queue);

        enable_interrupts();
        return(0);
    }

    /* receive interrupt */
    if (csr & LE_RINT)
    {
        /* clear interrupt */
        csr = LE_RINT | LE_INEA;

    /* process received frame */
    lance_receive();
        enable_interrupts();
        return();
    }
}
```


user
Send(.....)
1
socket
so
tcp
2
ip
3
4
Eth_dd
BD
Interrupt
handler
10
csr
0xffff3579h
5
5
9
5
6
csr
sco
int
CPU
8
Network adaptor
7
Network link

References

Main references

- W. Stallings, «Operating Systems», Chapter 17, 6th edition (freely available at https://app.box.com/s/mbh0v0f6nx)
- L. Peterson & B. Davie, «Computer Networks: A systems approach», 3rd edition, pp. 137-144

Further references

- A. Tanenbaum & D. Wetherall, «*Computer Networks*»
- W.R. Stevens, «Unix Network Programming»