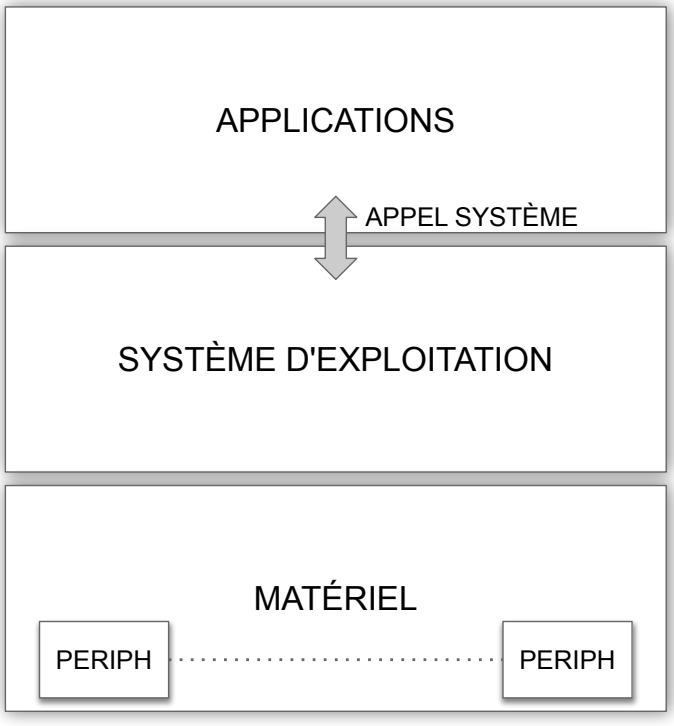
Modèle Client-Serveur

Module IOC – MU4IN109 – 2020fev

Franck Wajsbürt - François Pécheux

3 couches



Appels Système Fondamentaux (user)

Dans UNIX, « tout est fichier »

⇨ Un périphérique est un fichier

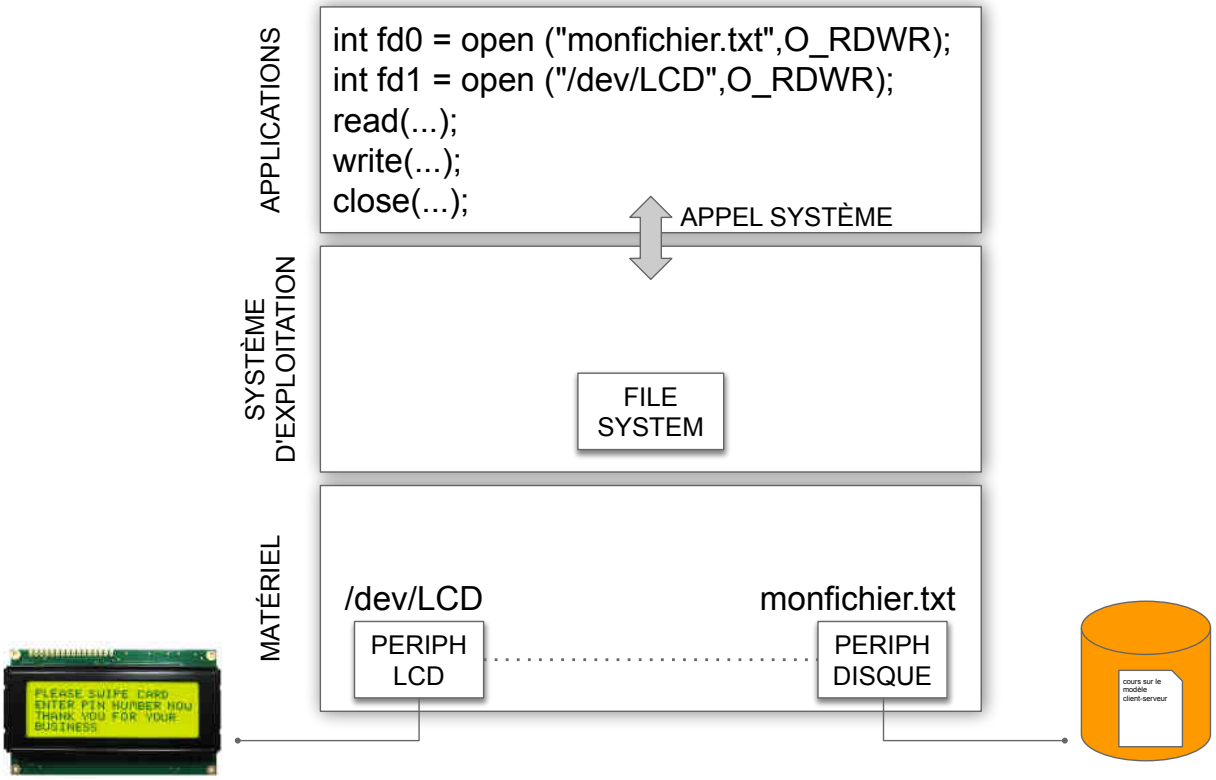
```
int fd; char * pathname ; int flags;
char buffer[100]; int len, actuel_len;
```

- `fd = open( pathname, flags );`
- `actual_len = read( fd, buffer, len );`
- `actual_len = write( fd, buffer, len );`
- `close( fd);`

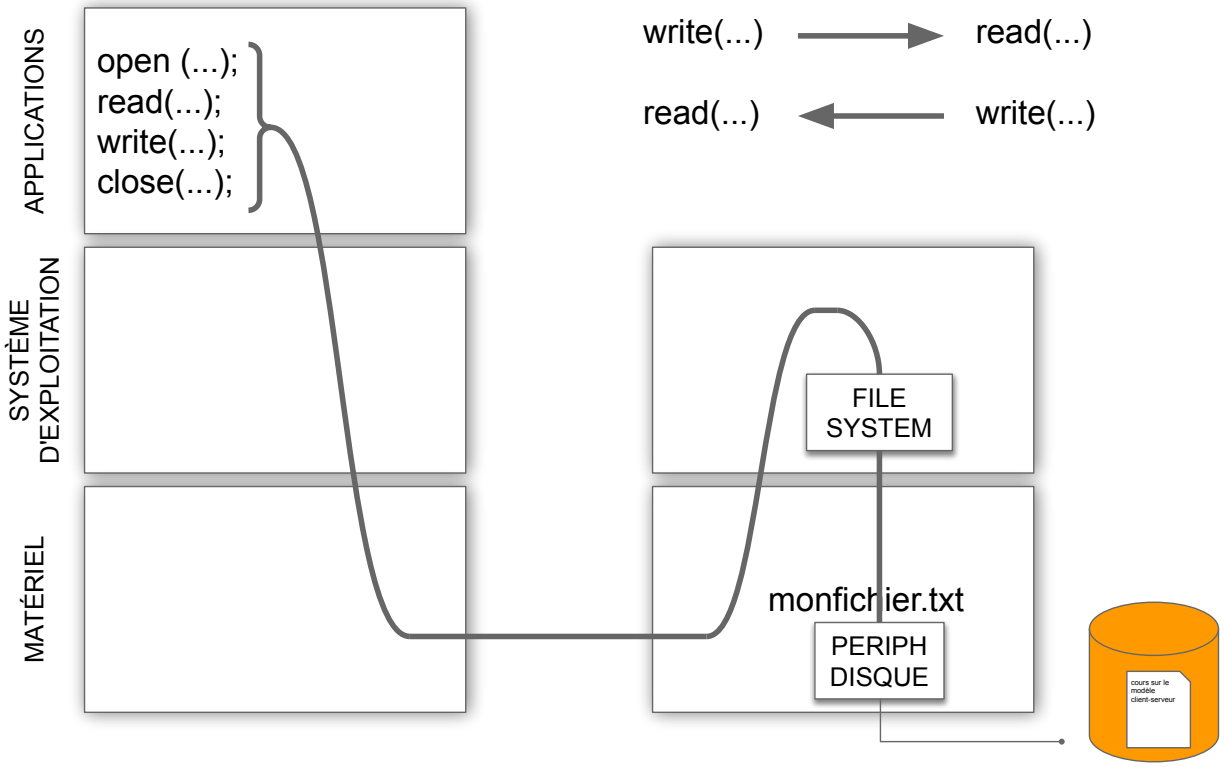
longueur max

longueur exacte

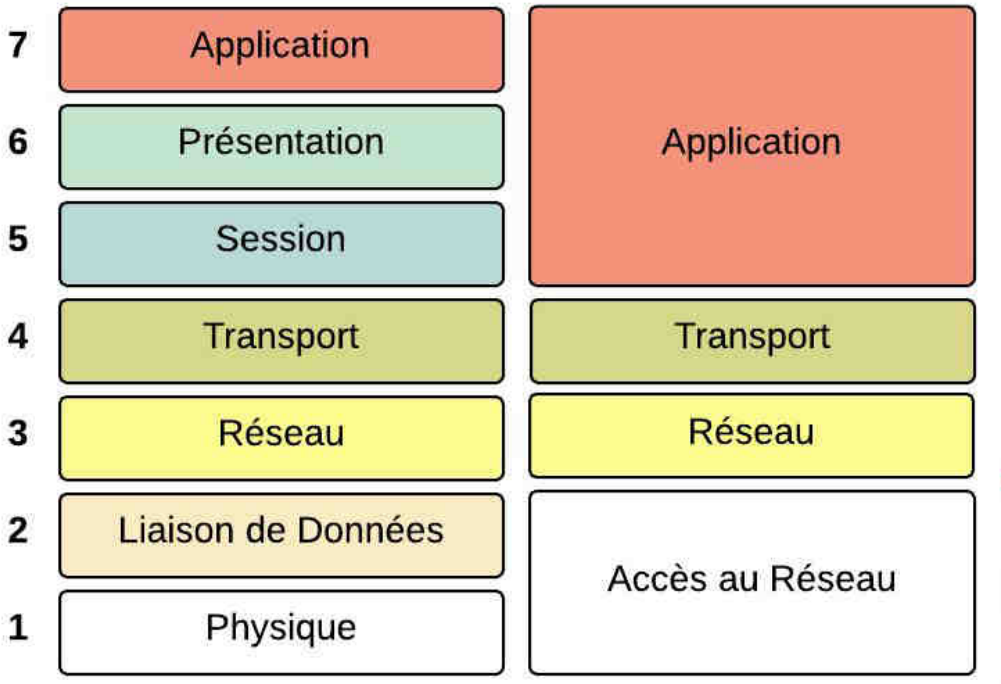
Accès Fichier



Accès Fichier

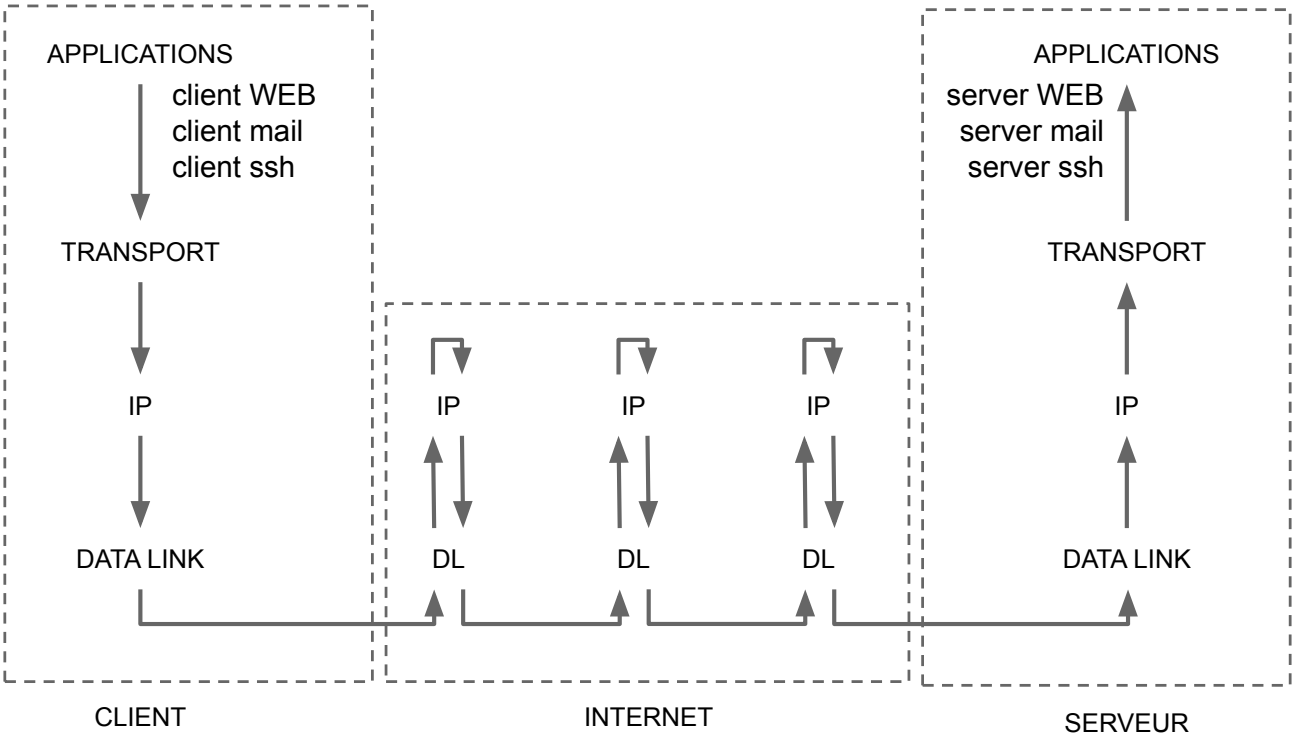


Modèle OSI – Modèle TCP/IP (unix)



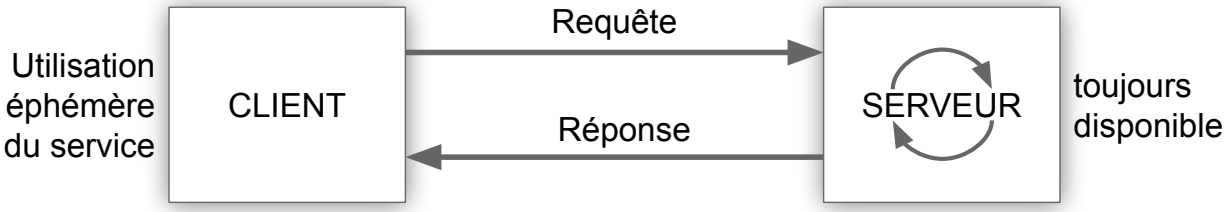
https://juleshuynhvan.business.blog/2017/02/20/modele-osi-modele-tcpip/

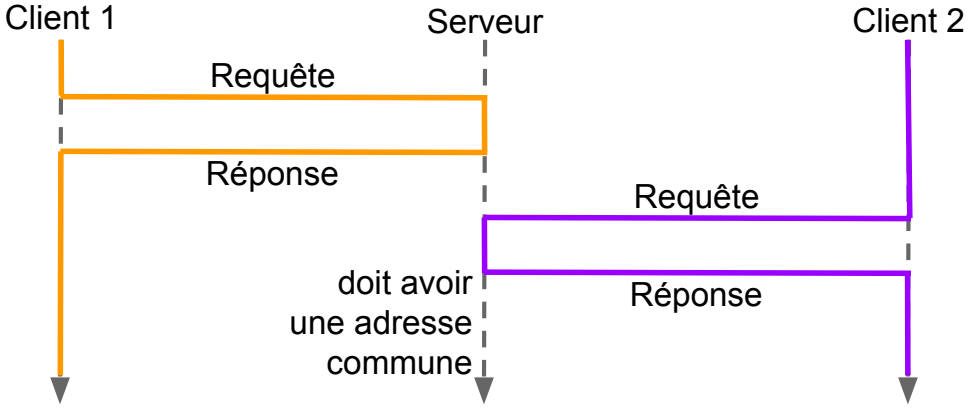
Echange Client – Serveur



SERVICES

Un serveur **rend** un service
Un client **demande** un service





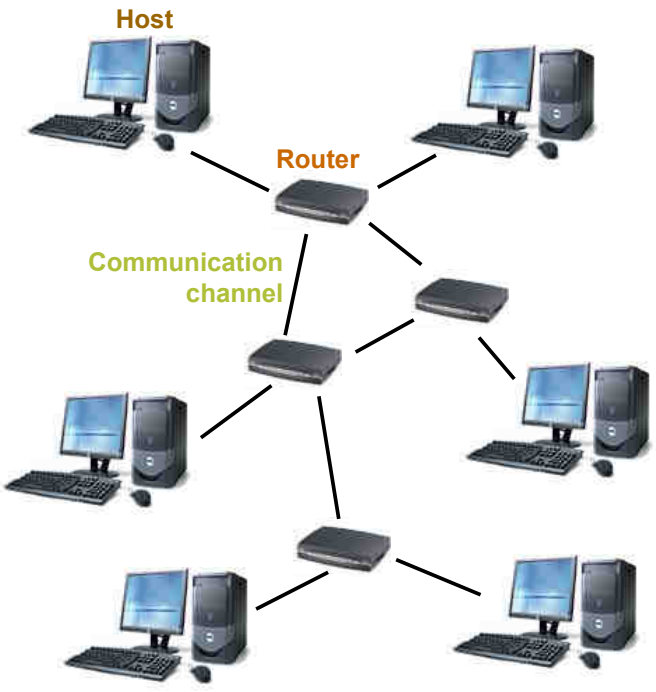
Introduction to Sockets Programming in C using TCP/IP

Professor: Panagiota Fatourou
TA: Eleftherios Kosmas
CSD - May 2012

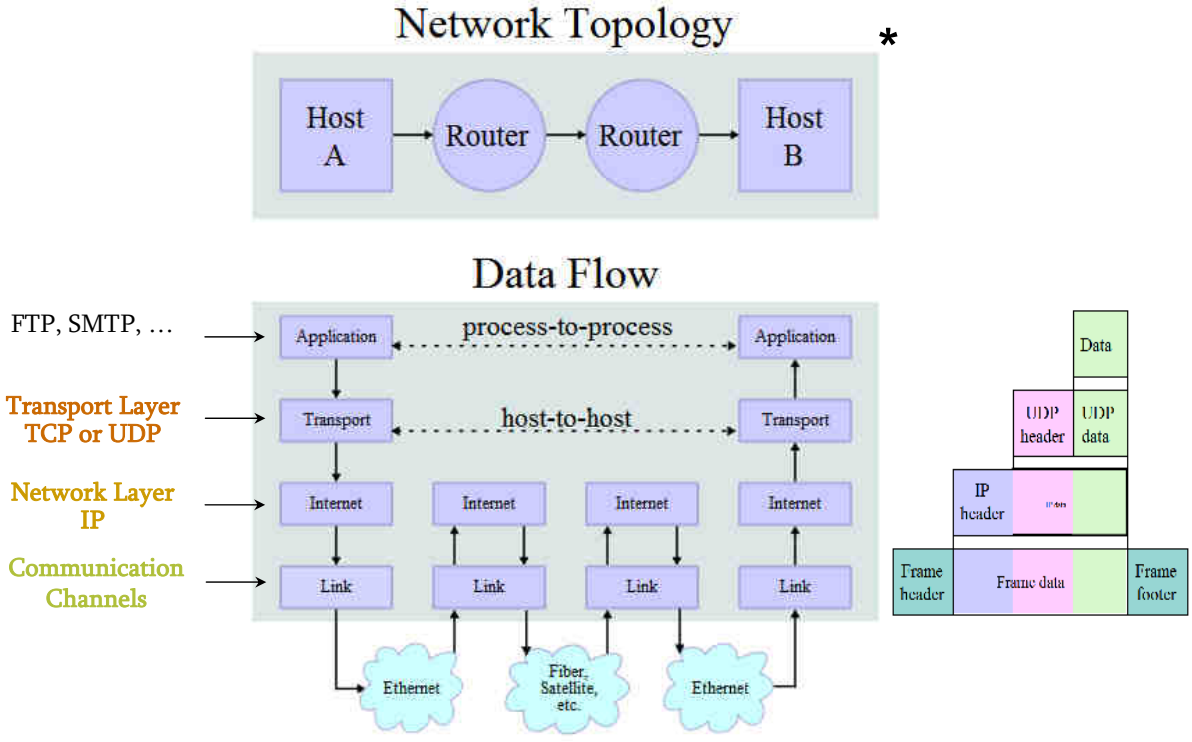
Introduction

- Computer Network
  - hosts, routers, communication channels
- Hosts run applications
- Routers forward information
- Packets: sequence of bytes
  - contain control information
  - e.g. destination host
- Protocol is an agreement
  - meaning of packets
  - structure and size of packets

  e.g. Hypertext Transfer Protocol (HTTP)



TCP/IP



* image is taken from "http://en.wikipedia.org/wiki/TCP/IP_model"

Internet Protocol (IP)

- provides a **datagram** service
  - packets are handled and delivered independently
- **best-effort** protocol
  - may loose, reorder or duplicate packets

- each packet must contain an **IP address** of its destination

TCP vs UDP

- Both use **port numbers**
  - application-specific construct serving as a communication endpoint
  - 16-bit unsigned integer, thus ranging from 0 to 65535
  - ☞ to provide **end-to-end** transport
- UDP: User Datagram Protocol
  - no acknowledgements
  - no retransmissions
  - out of order, duplicates possible
  - connectionless, i.e., app indicates destination for each packet
- TCP: Transmission Control Protocol
  - reliable **byte-stream channel** (in order, all arrive, no duplicates)
    - similar to file I/O
  - flow control
  - connection-oriented
  - bidirectional

TCP vs UDP

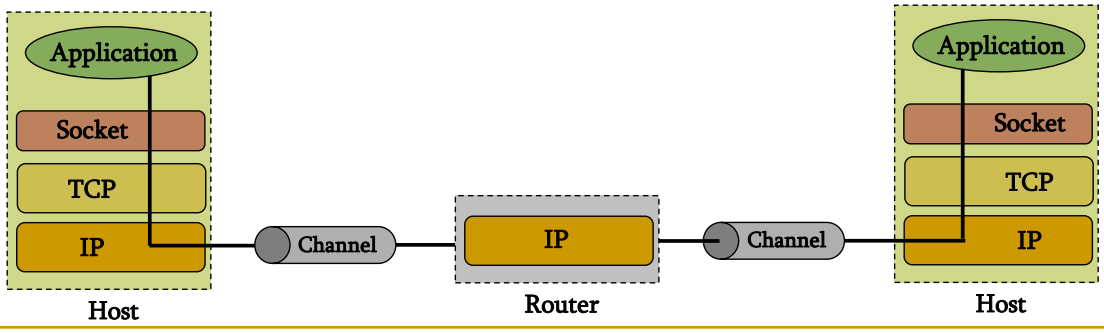
- TCP is used for services with a large data capacity, and a persistent connection
- UDP is more commonly used for quick lookups, and single use query-reply actions.

- Some common examples of TCP and UDP with their default ports:

DNS lookup	UDP	53
FTP	TCP	21
HTTP	TCP	80
POP3	TCP	110
Telnet	TCP	23

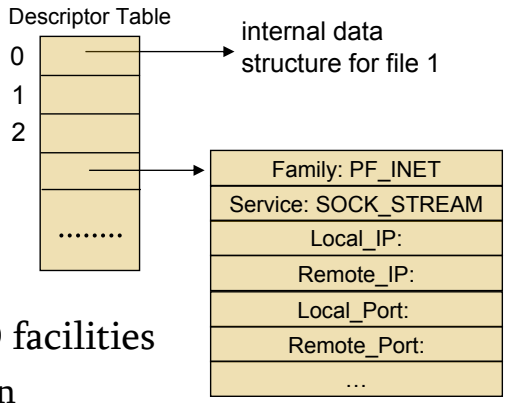
Berkley Sockets

- Universally known as Sockets
- It is an abstraction through which an application may send and receive data
- Provide generic access to interprocess communication services
  - e.g. IPX/SPX, Appletalk, TCP/IP
- Standard API for networking

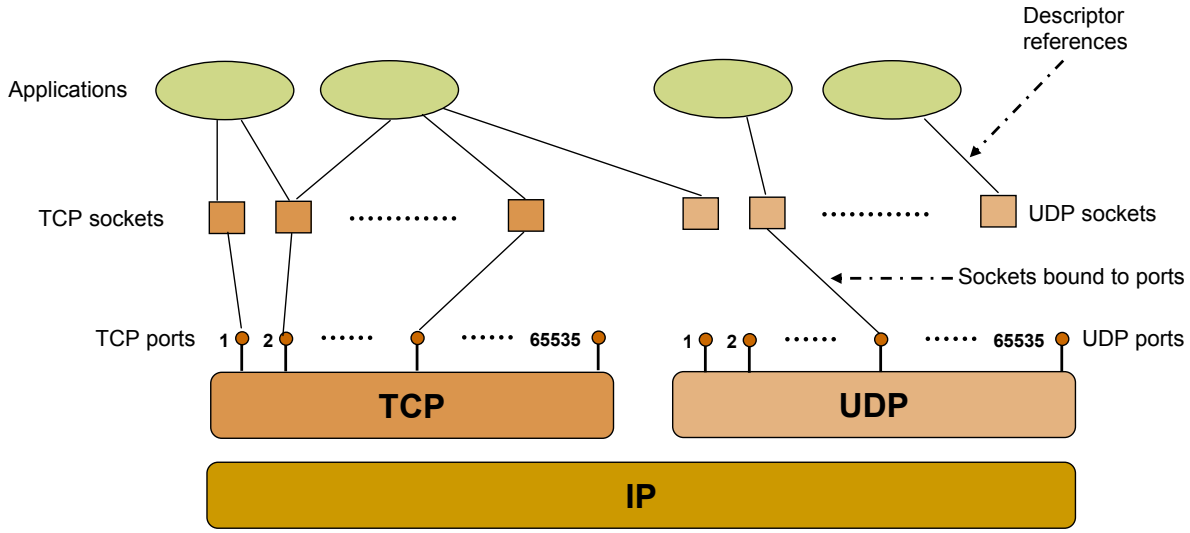


Sockets

- Uniquely identified by
  - an internet address
  - an end-to-end protocol (e.g. TCP or UDP)
  - a port number
- Two types of (TCP/IP) sockets
  - Stream sockets (e.g. uses TCP)
    - provide reliable byte-stream service
  - Datagram sockets (e.g. uses UDP)
    - provide best-effort datagram service
    - messages up to 65.500 bytes
- Socket extend the convectional UNIX I/O facilities
  - file descriptors for network communication
  - extended the read and write system calls



Sockets



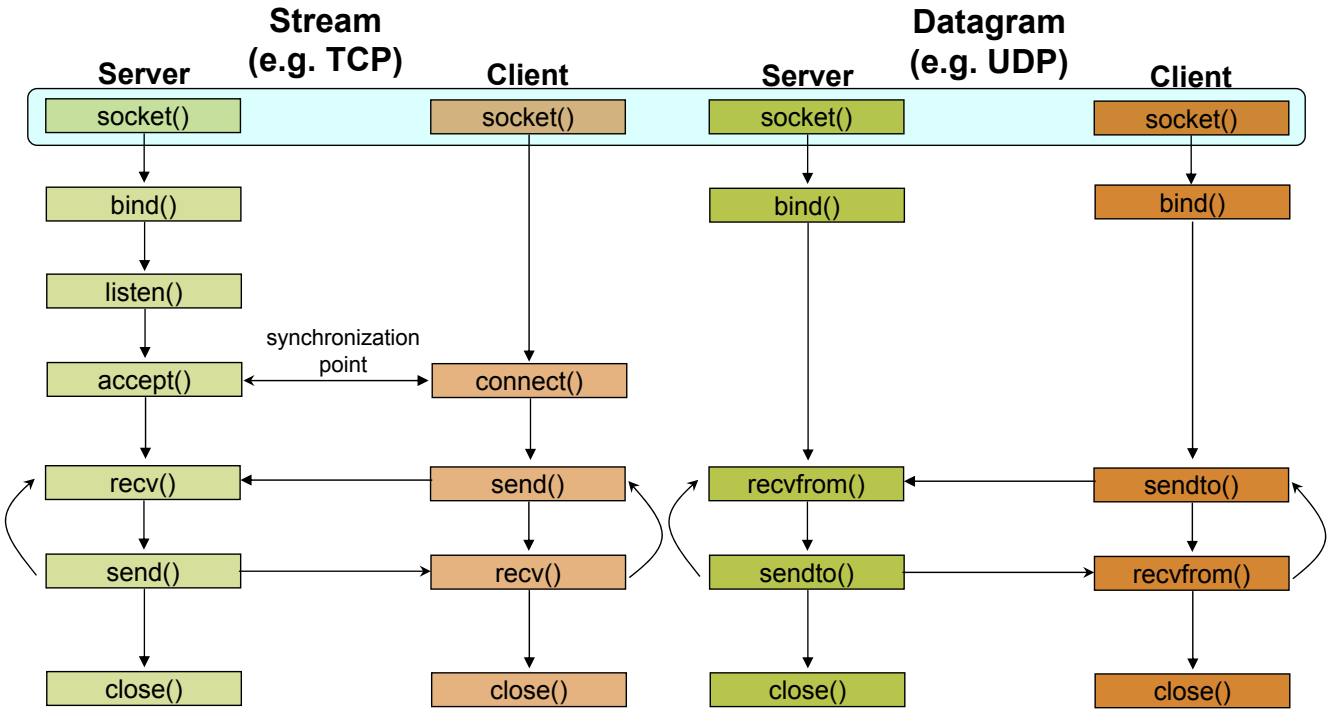
Client-Server communication

- Server
  - passively waits for and responds to clients
  - passive socket
- Client
  - initiates the communication
  - must know the address and the port of the server
  - active socket

Sockets - Procedures

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Client - Server Communication - Unix



Socket creation in C: `socket()`

- `int sockid = socket(family, type, protocol);`
  - sockid: socket descriptor, an integer (like a file-handle)
  - family: integer, communication domain, e.g.,
    - PF_INET, IPv4 protocols, Internet addresses (typically used)
    - PF_UNIX, Local communication, File addresses
  - type: communication type
    - SOCK_STREAM - reliable, 2-way, connection-based service
    - SOCK_DGRAM - unreliable, connectionless, messages of maximum length
  - protocol: specifies protocol
    - IPPROTO_TCP IPPROTO_UDP
    - usually set to 0 (i.e., use default protocol)
  - upon failure returns -1

☞ NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

Assign address to socket: `bind()`

- associates and reserves a port for use by the socket

- `int status = bind(sockid, &addrport, size);`
  - sockid: integer, socket descriptor
  - addrport: struct sockaddr, the (IP) address and port of the machine
    - for TCP/IP server, internet address is usually set to INADDR_ANY, i.e., chooses any incoming interface
  - size: the size (in bytes) of the addrport structure
  - status: upon failure -1 is returned

bind() - Example with TCP

```
int sockid;
struct sockaddr_in addrport;
sockid = socket(PF_INET, SOCK_STREAM, 0);

addrport.sin_family = AF_INET;
addrport.sin_port = htons(5100);
addrport.sin_addr.s_addr = htonl(INADDR_ANY);
if(bind(sockid, (struct sockaddr *) &addrport, sizeof(addrport))!= -1) {
    …}
```


Assign address to socket: bind()

- Instructs TCP protocol implementation to listen for connections
- `int status = listen(sockid, queueLimit);`
  - sockid: integer, socket descriptor
  - queuelen: integer, # of active participants that can "wait" for a connection
  - status: 0 if listening, -1 if error
- `listen()` is **non-blocking**: returns immediately
- The listening socket (sockid)
  - is never used for sending and receiving
  - is used by the server only as a way to get new sockets

Establish Connection: `connect()`

- The client establishes a connection with the server by calling `connect()`

- `int status = connect(sockid, &foreignAddr, addrlen);`
  - sockid: integer, socket to be used in connection
  - foreignAddr: struct sockaddr: address of the passive participant
  - addrlen: integer, sizeof(name)
  - status: 0 if successful connect, -1 otherwise
- `connect()` is **blocking**

Incoming Connection: `accept()`

- The server gets a socket for an incoming client connection by calling `accept()`
- `int s = accept(sockid, &clientAddr, &addrLen);`
  - s: integer, the new socket (used for data-transfer)
  - sockid: integer, the orig. socket (being listened on)
  - clientAddr: struct sockaddr, address of the active participant
    - filled in upon return
  - addrLen: sizeof(clientAddr): value/result parameter
    - must be set appropriately before call
    - adjusted upon return
- `accept()`
  - is **blocking**: waits for connection before returning
  - dequeues the next connection on the queue for socket (sockid)

Exchanging data with stream socket

- `int count = send(sockid, msg, msgLen, flags);`
  - msg: const void[], message to be transmitted
  - msgLen: integer, length of message (in bytes) to transmit
  - flags: integer, special options, usually just 0
  - count: # bytes transmitted (-1 if error)
- `int count = recv(sockid, recvBuf, bufLen, flags);`
  - recvBuf: void[], stores received bytes
  - bufLen: # bytes received
  - flags: integer, special options, usually just 0
  - count: # bytes received (-1 if error)
- Calls are **blocking**
  - returns only after data is sent / received

Socket close in C: `close()`

- When finished using a socket, the socket should be closed
- `status = close(sockid);`
  - **sockid**: the file descriptor (socket being closed)
  - **status**: 0 if successful, -1 if error
- Closing a socket
  - closes a connection (for stream socket)
  - frees up the port used by the socket

Example - Echo using stream socket

```
/* Create socket for incoming connections */
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
     DieWithError("socket() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. **Create a TCP socket**
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

Example - Echo using stream socket

```
echoServAddr.sin_family = AF_INET;                  /* Internet address family */
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);   /* Any incoming interface */
echoServAddr.sin_port = htons(echoServPort);        /* Local port */

if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("bind() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. **Assign a port to socket**
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

Example - Echo using stream socket

```
/* Mark the socket so it will listen for incoming connections */
if (listen(servSock, MAXPENDING) < 0)
      DieWithError("listen() failed");
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. **Set socket to listen**
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

Example - Echo using stream socket

```
for (;;) /* Run forever */
{
  clntLen = sizeof(echoClntAddr);

  if ((clientSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen))<0)
      DieWithError("accept() failed");
  ...
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. **Accept new connection**
   b. Communicate
   c. Close the connection

Example - Echo using stream socket

Server is now blocked waiting for connection from a client

...

A client decides to talk to the server

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. **Accept new connection**
   b. Communicate
   c. Close the connection

Example - Echo using stream socket

```
/* Create a reliable, stream socket using TCP */
if ((clientSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
     DieWithError("socket() failed");
```

Client

1. **Create a TCP socket**
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. **Accept new connection**
   b. Communicate
   c. Close the connection

Example - Echo using stream socket

```
echoServAddr.sin_family = AF_INET;                    /* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(echoservIP); /* Server IP address*/
echoServAddr.sin_port = htons(echoServPort);          /* Server port */

if (connect(clientSock, (struct sockaddr *) &echoServAddr,
                        sizeof(echoServAddr)) < 0)
    DieWithError("connect() failed");
```

Client

1. Create a TCP socket
2. **Establish connection**
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. **Accept new connection**
   b. Communicate
   c. Close the connection

Example - Echo using stream socket

Server's accept procedure in now unblocked and returns client's socket

```
for (;;) /* Run forever */
{
  clntLen = sizeof(echoClntAddr);

  if ((clientSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen))<0)
     DieWithError("accept() failed");
  ...
```

Client

1. Create a TCP socket
2. **Establish connection**
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. **Accept new connection**
   b. Communicate
   c. Close the connection

Example - Echo using stream socket

```
echoStringLen = strlen(echoString);     /* Determine input length */

/* Send the string to the server */
if (send(clientSock, echoString, echoStringLen, 0) != echoStringLen)
     DieWithError("send() sent a different number of bytes than expected");
```

Client

1. Create a TCP socket
2. Establish connection
3. **Communicate**
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. **Accept new connection**
   b. Communicate
   c. Close the connection

Example - Echo using stream socket

```
/* Receive message from client */
if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
    DieWithError("recv() failed");
/* Send received string and receive again until end of transmission */
while (recvMsgSize > 0) {  /* zero indicates end of transmission */
   if (send(clientSocket, echobuffer, recvMsgSize, 0) != recvMsgSize)
      DieWithError("send() failed");
   if ((recvMsgSize = recv(clientSocket, echoBuffer, RECVBUFSIZE, 0)) < 0)
      DieWithError("recv() failed");
}
```

Client

1. Create a TCP socket
2. Establish connection
3. **Communicate**
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. **Communicate**
   c. Close the connection

Example - Echo using stream socket

Similarly, the client receives the data from the server

Client

1. Create a TCP socket
2. Establish connection
3. **Communicate**
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. **Communicate**
   c. Close the connection

Example - Echo using stream socket

```
close(clientSock);
```

```
close(clientSock);
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. **Close the connection**

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. **Close the connection**

Example - Echo using stream socket

Server is now blocked waiting for connection from a client ...

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. **Accept new connection**
   b. Communicate
   c. Close the connection

En TME

Programmer une application de vote client-serveur et...