UNIX Network Programming

Intro to BSD Sockets

Programming with sockets in the IPv4 and IPv6 domains

Catiuscia Melle and Franco Tommasi - SNP 2021-2022

References

- ***UNIX® Network Programming Volume 1, Third Edition: The Sockets Networking API***, *W. R. Stevens*

  `http://www.unpbook.com/`

- **The Open Group Base Specifications Issue 7**

  `http://pubs.opengroup.org/onlinepubs/9699919799/`

- **The Linux Programming Interface - A Linux and UNIX® System Programming Handbook**, *M. Kerrisk*

  `http://man7.org`

- **Beej's Guide to Network Programming:**

  `http://beej.us/guide/bgnet/`

- **TCP/IP Sockets in C: Practical Guide for Programmers, 2nd Ed**, M. J. Donahoo and K. L. Calvert

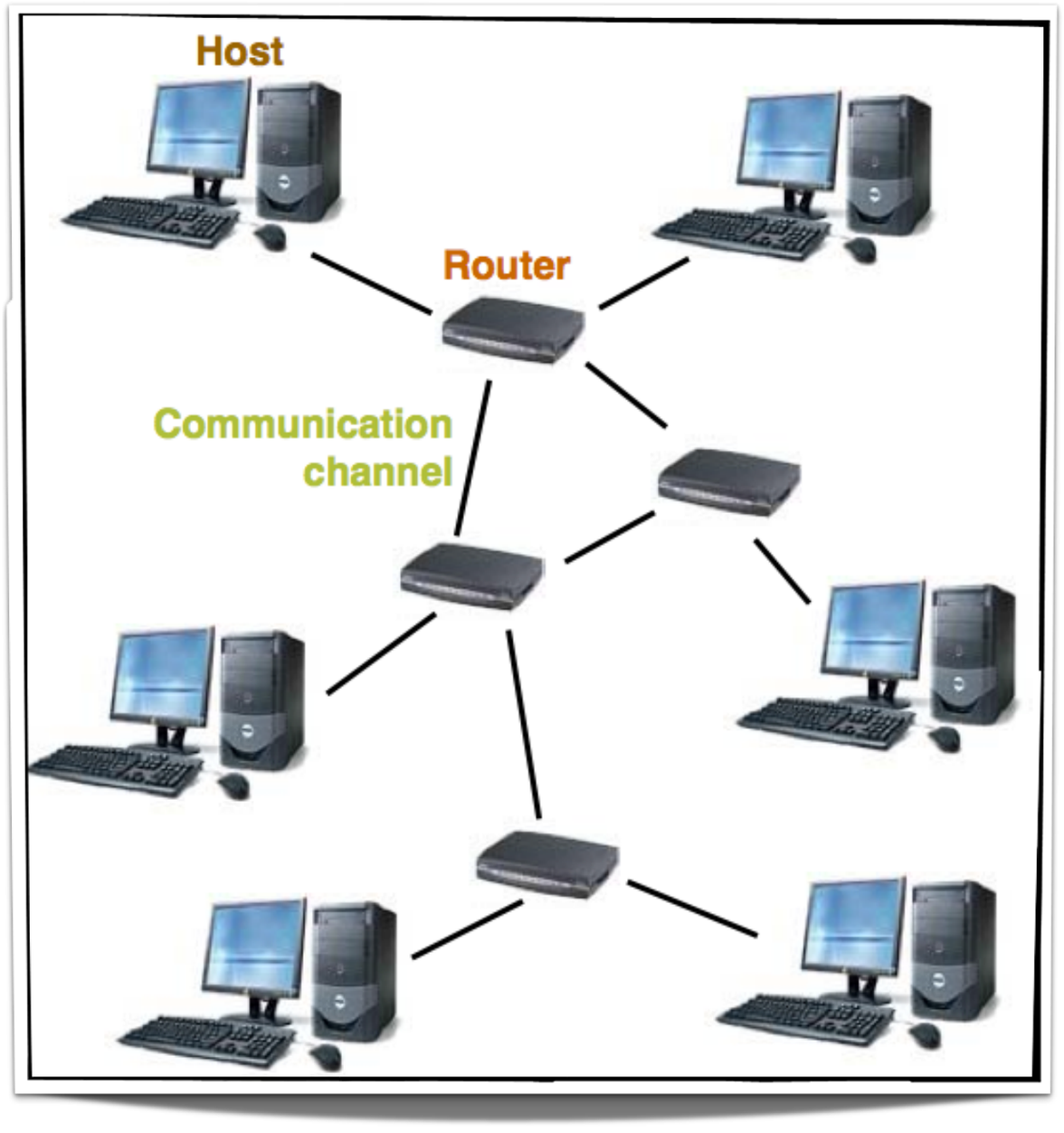
  `Cercare "TCP IP Sockets in C". Su GitHub è disponibile il PDF`

- **GAPIL**: Guida alla Programmazione in Linux

  `gapil.gnulinux.it/`

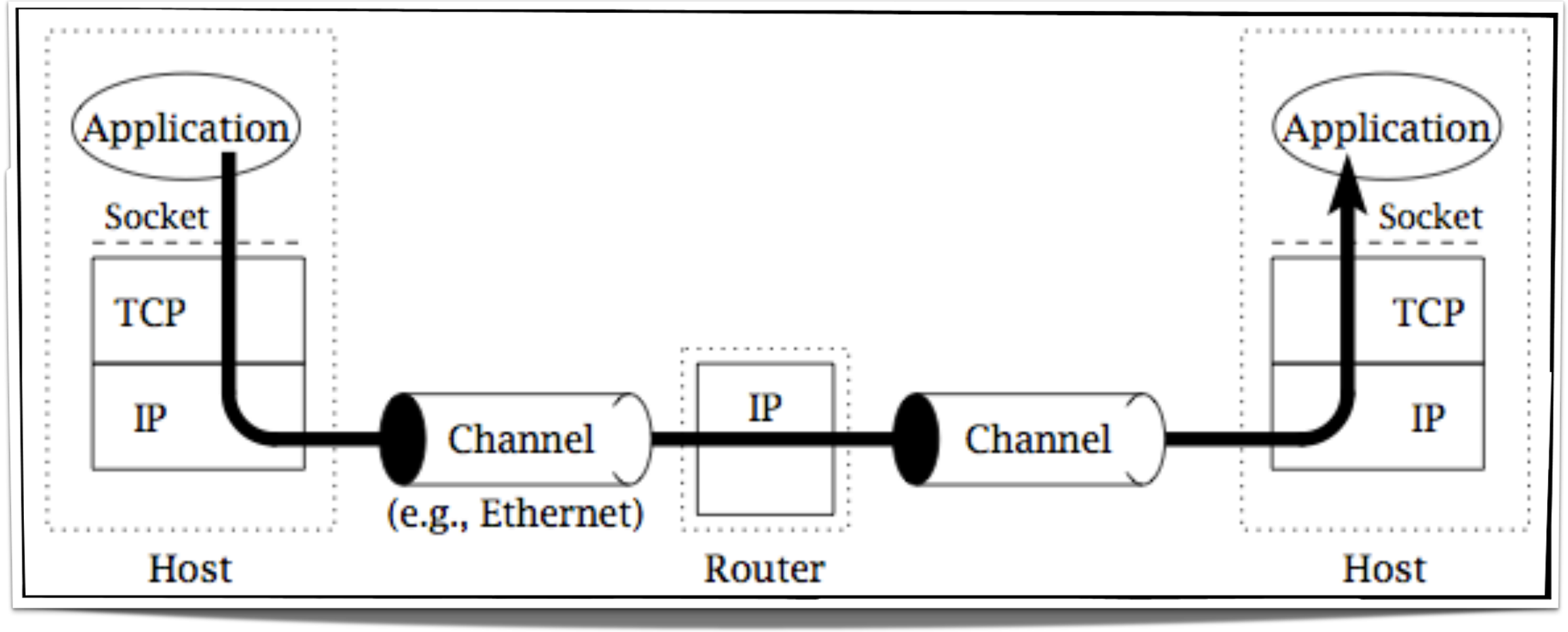
Sockets Programming

- Sockets are a method of IPC
- They allow data to be exchanged between applications:
  - either on the ***same host***
  - or on different host connected by a ***network***
- The first implementation of the Sockets API (**A**pplication **P**rogramming **I**nterface) appeared in ***4.2BSD*** in *1983*



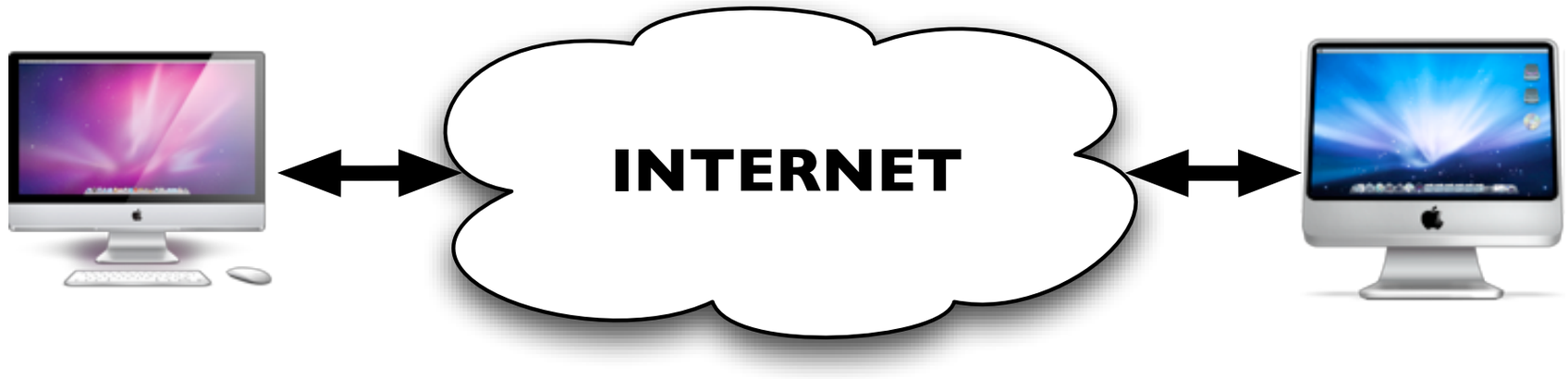
Sockets Programming

- Sockets extend the conventional UNIX I/O facilities:
  - in Unix *"everything is a file"*
  - *file descriptors* for network communication
  - extends the *read* and *write* system calls



Sockets Programming and the Internet

- BSD Sockets are the cornerstone of a form of **interprocess communication.** Such **IPC** take place among **processes** on **remote hosts** connected to the **Internet**
  - (that is) networks based on the ***TCP/IP protocol suite***

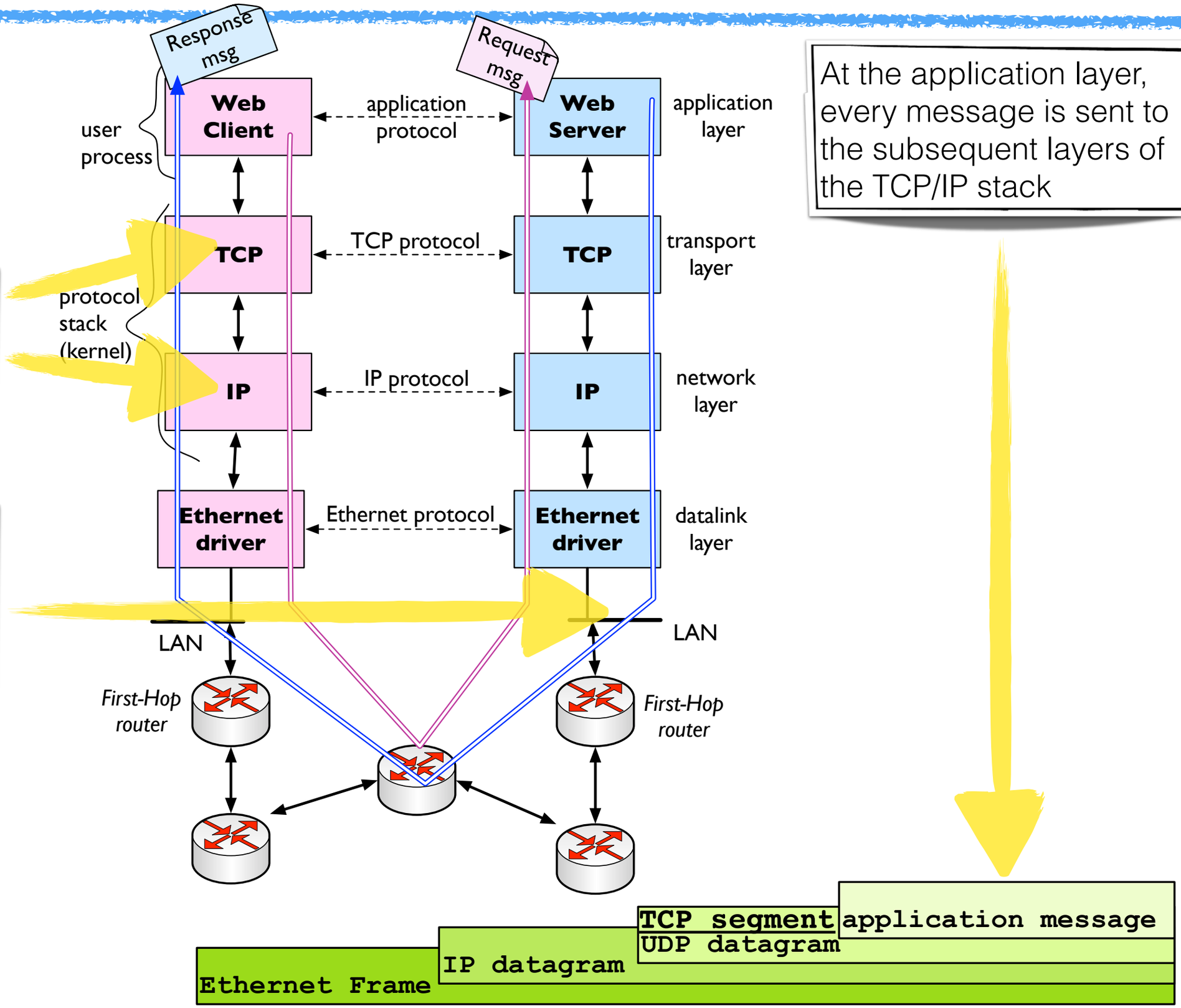


- In a TCP/IP based network, each host is uniquely identified by its IP address.
- We will consider:
  - **IPv4 domain**
  - **IPv6 domain**

Definition of a communication protocol

- A *network application* is made up of *different parts*, deployed on several *different hosts*
  - The different parts exchange information: to allow such interaction, the application must define a common *form of communication*, the **protocol**
- A protocol describes **communication rules**, defining:
  - the possible *interactions* among the *applications*
    - *who **starts** the communication,*
    - *who has to **wait** for a response, …*
  - the ***message format***
  - the ***data*** to be exchanged
- Example: the *HTTP* protocol is based on the **client-server model:**
  - only the *Web client* (browser) starts the communication, sending its **requests** to the server
  - the *Web server* (usually a daemon) **replies** to received requests, sending the desired information

Encapsulation



Every layer provide an abstraction of the service

The data is exchanged only by the datalink layer, over the physical link

At the application layer, every message is sent to the subsequent layers of the TCP/IP stack

Traffic sniffers

- While dealing with networking applications, it's important to fully understand what happens to our data when they traverse the network.
- An absolutely necessary tool to perform such kind of analysis is a *traffic sniffer*:

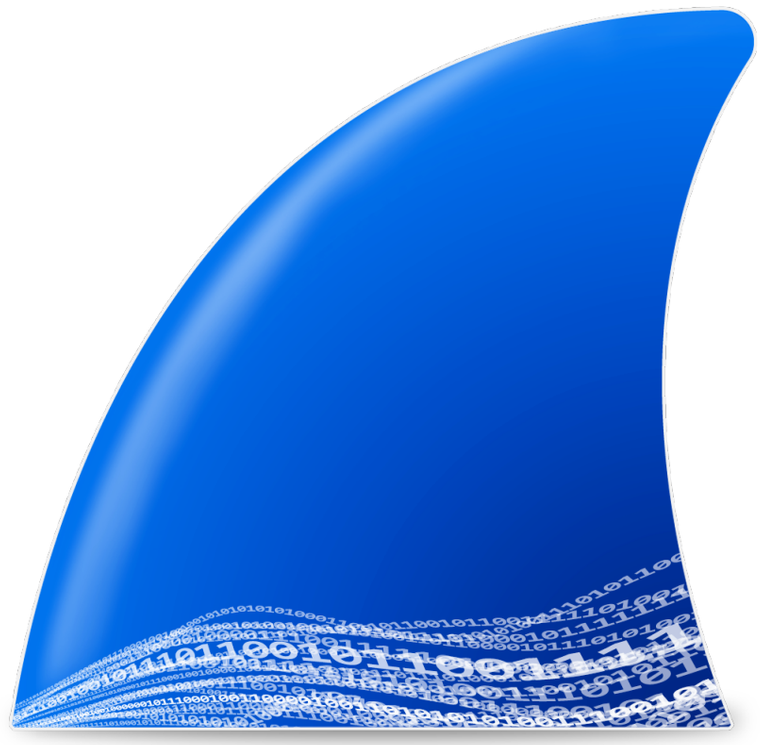
  `www.tcpdump.org`

- is the URL of the **tcpdump** project and **libpcap** library
  - *tcpdump* is a command-line traffic sniffer
  - *libpcap* its the C/C++ portable library for network traffic capture
    - libpcap APIs let a sniffer to intercept and make a copy of all the traffic that traverse a Network Interface (*NIC*)

Wireshark

`http://www.wireshark.org`

- is a multi-platform network protocol analyser, with a graphic front-end
  - the world's foremost sniffer
- It implements many features that let's examine captured packets.
- After the capture, Wireshark features let's post-process the traffic to:
  - filter only desired conversations,
  - obtain statistical analyses
  - determine loss rate
  - bytes/second exchanged
  - ...

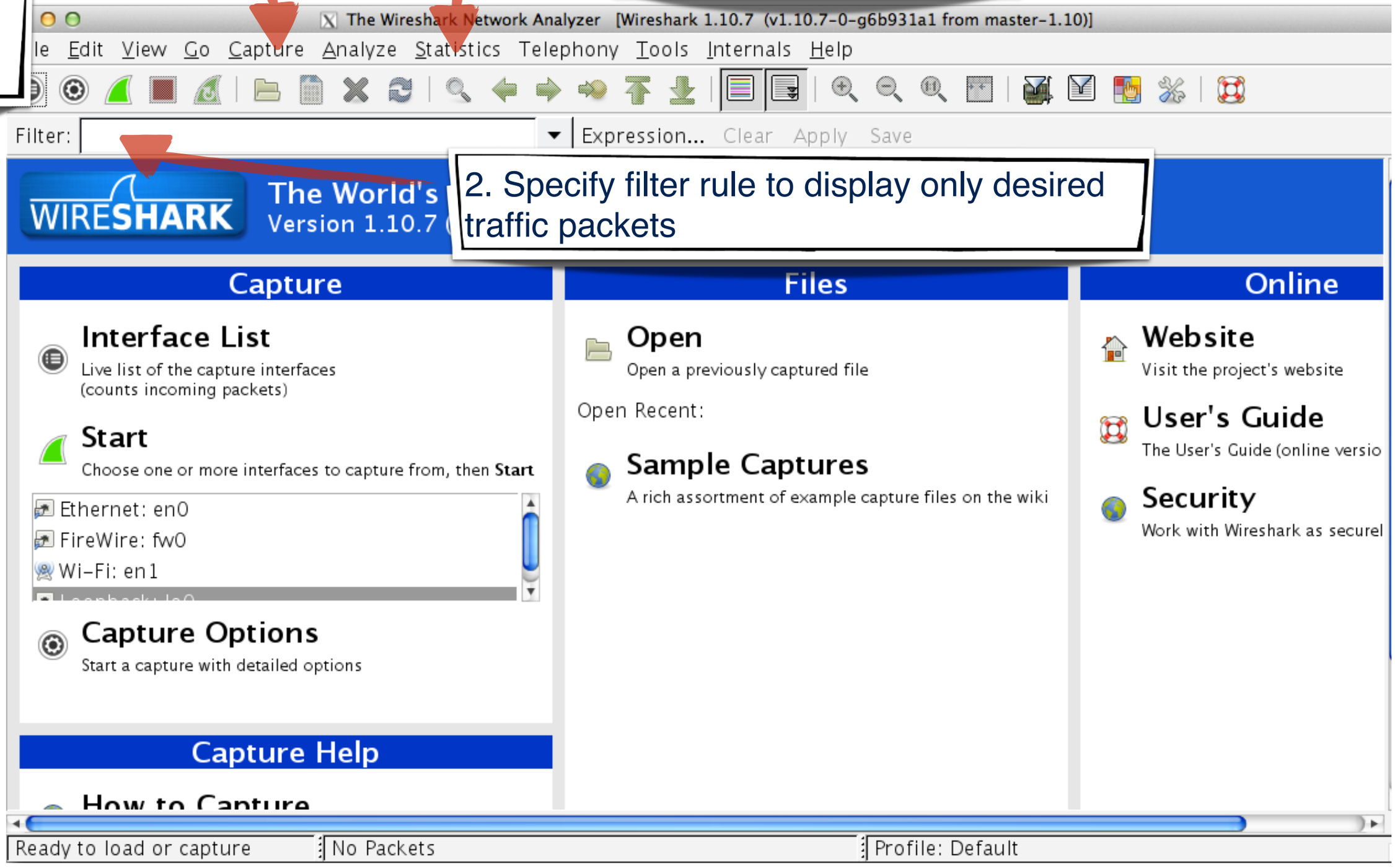


Wireshark

1. Specify the interface to monitor

3. Post-processing on traffic

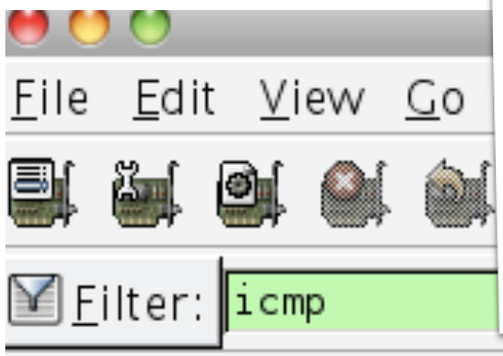
2. Specify filter rule to display only desired traffic packets



ping and traceroute

- **ping** command, based on the *ICMP protocol*, is used to estimate the **RTT** between two nodes on the network, using the *ICMP Request/Reply messages*

```
$ping 193.204.77.77
PING 193.204.77.77 (193.204.77.77): 56 data bytes
64 bytes from 193.204.77.77: icmp_seq=0 ttl=48 time=126.477 ms
64 bytes from 193.204.77.77: icmp_seq=1 ttl=48 time=134.870 ms
64 bytes from 193.204.77.77: icmp_seq=2 ttl=48 time=124.623 ms
64 bytes from 193.204.77.77: icmp_seq=3 ttl=48 time=134.719 ms
64 bytes from 193.204.77.77: icmp_seq=4 ttl=48 time=125.470 ms
^C
--- 193.204.77.77 ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 124.623/129.232/134.870/4.580 ms
```



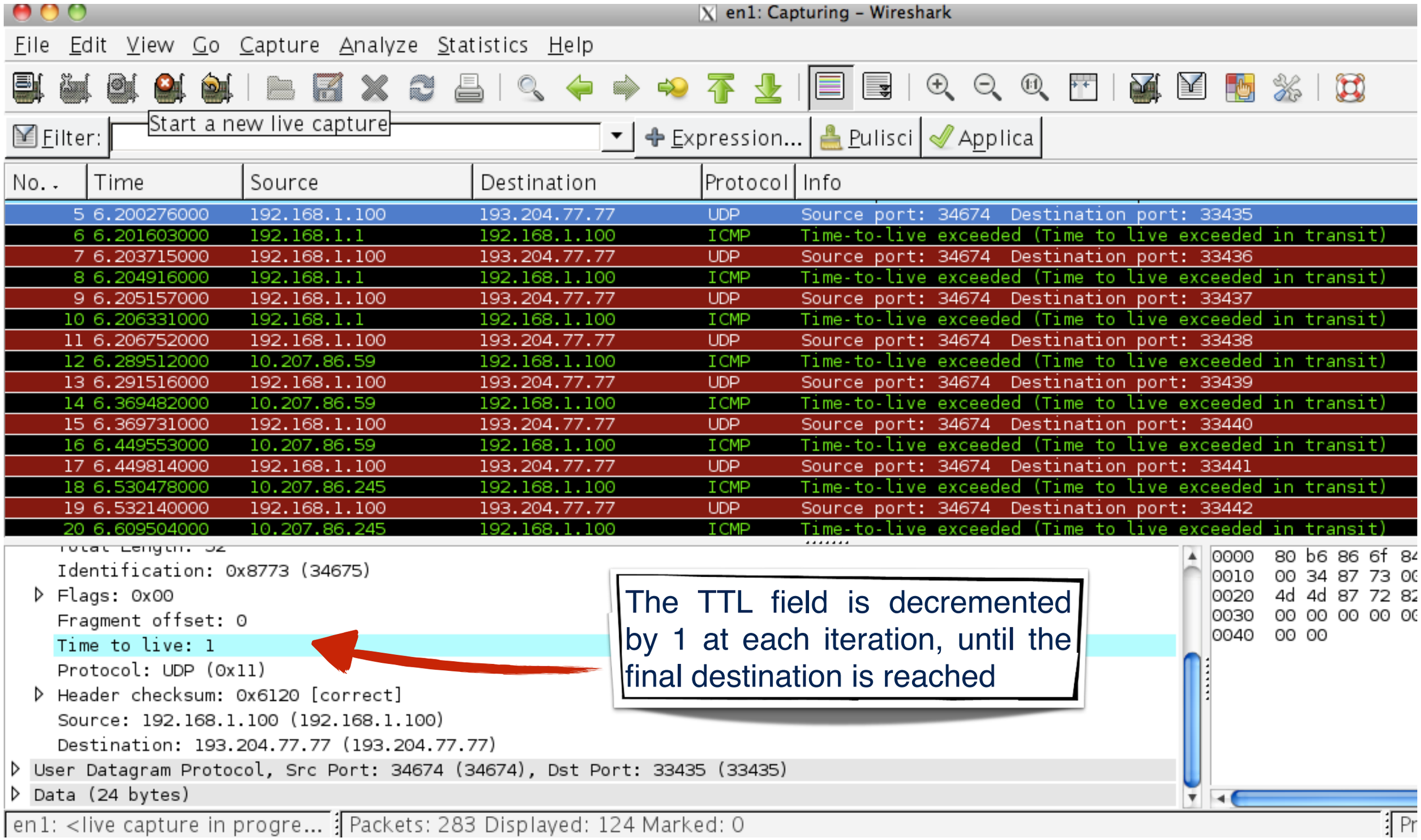
No.	Time	Source	Destination	Protocol	Info
9	2.663110000	192.168.1.100	193.204.77.77	ICMP	Echo (ping) request
10	2.789509000	193.204.77.77	192.168.1.100	ICMP	Echo (ping) reply
11	3.663398000	192.168.1.100	193.204.77.77	ICMP	Echo (ping) request
12	3.798138000	193.204.77.77	192.168.1.100	ICMP	Echo (ping) reply
13	4.663577000	192.168.1.100	193.204.77.77	ICMP	Echo (ping) request
14	4.788078000	193.204.77.77	192.168.1.100	ICMP	Echo (ping) reply
15	5.663834000	192.168.1.100	193.204.77.77	ICMP	Echo (ping) request
16	5.798381000	193.204.77.77	192.168.1.100	ICMP	Echo (ping) reply
17	6.664084000	192.168.1.100	193.204.77.77	ICMP	Echo (ping) request
18	6.789395000	193.204.77.77	192.168.1.100	ICMP	Echo (ping) reply

ping and traceroute

- **traceroute** command let's determine the *route* traversed by packets delivered between two nodes, based on the ***TTL*** field of the IP protocol header.

```
$traceroute 193.204.77.77
traceroute to 193.204.77.77 (193.204.77.77), 64 hops max, 52 byte packets
 1  192.168.1.1 (192.168.1.1)  1.800 ms  1.289 ms  1.334 ms
 2  10.207.86.59 (10.207.86.59)  82.955 ms  78.167 ms  80.020 ms
 3  10.207.86.245 (10.207.86.245)  80.858 ms  77.541 ms  79.829 ms
 4  217.141.109.208 (217.141.109.208)  79.994 ms  79.139 ms  80.714 ms
 5  172.17.5.157 (172.17.5.157)  89.378 ms  88.162 ms  90.092 ms
 6  172.17.8.70 (172.17.8.70)  99.892 ms  98.908 ms  100.218 ms
 7  r-rm83-vl4.opb.interbusiness.it (151.99.29.203)  99.787 ms  98.339 ms  99.897 ms
 8  172.17.5.206 (172.17.5.206)  99.915 ms  98.860 ms  99.935 ms
 9  garr-nap.namex.it (193.201.28.15)  99.962 ms  98.352 ms  99.819 ms
10  90.147.94.61 (90.147.94.61)  99.805 ms  98.422 ms  100.037 ms
11  rt1-bo1-rt-rm2.rm2.garr.net (193.206.141.5)  109.753 ms  109.342 ms  110.386 ms
12  rt1-bo1-rt-ba1.ba1.garr.net (193.206.134.78)  119.365 ms  117.892 ms  109.904 ms
13  rt-ba1-ru-unile.ba1.garr.net (193.206.137.110)  120.079 ms  117.951 ms  120.185 ms
14  cisco7609.unile.it (193.204.86.9)  119.723 ms  118.383 ms  120.259 ms
15  4006.to.7609.unile.it (193.204.71.114)  118.205 ms  118.002 ms  120.013 ms
16  stecca.to.4006.unile.it (193.204.71.254)  130.147 ms  127.961 ms  119.848 ms
17  193.204.77.77 (193.204.77.77)  129.819 ms  129.946 ms  128.268 ms
```

ping and traceroute



The TTL field is decremented by 1 at each iteration, until the final destination is reached

ifconfig

real interface

```
kat@hplinux2:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:a6:a7:08
          inet addr:193.204.77.157  Bcast:193.204.77.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fea6:a708/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:27115189 errors:0 dropped:182435 overruns:0 frame:0
          TX packets:9875459 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4231205921 (4.2 GB)  TX bytes:2642874300 (2.6 GB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:203101 errors:0 dropped:0 overruns:0 frame:0
          TX packets:203101 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:20164337 (20.1 MB)  TX bytes:20164337 (20.1 MB)
```

virtual interface

- a similar output is provided by

```
netstat -i
```

netstat and lsof

- In this example, two processes (a TCP client and server) running on the same host, require resources allocated in the kernel, shown by the commands:

a TCP client connected to the server

the TCP Server

TCP-IPv4-v4 — TCPClient — 86×13

```
Kliis:TCP-IPv4-v4 katjusha$ ./TCPClient
TCP echo client, version 4
Try to call TCP echo server on localhost on port 43155
Insert the message to send
hello
prima di inviare
secondo invio
res = 6
Attendo echoed message
hello
Again?...[Ctrl+D per terminare]
ciao
```

TCP-IPv4-v4 — kat@hplinux2: ~ — TCPServer — 75×14 — ⌘2

```
Kliis:TCP-IPv4-v4 katjusha$ ./TCPServer
TCP echo server, version 4
Waiting for client connections on port 43155

        Managing connection from client '127.0.0.1:65184'
Received message: hello server
sent 13 bytes from hello server

Echo sent back con res = 13
Received message: hello server
sent 13 bytes from hello server
```

TCP-IPv4-v2 — bash — 118×19 — ⌘4

```
Kliis:TCP-IPv4-v2 katjusha$ netstat -a -p tcp | grep 43155
tcp4       0      0  localhost.43155        localhost.65195        ESTABLISHED
tcp4      52      0  localhost.65195        localhost.43155        ESTABLISHED
tcp4       0      0  *.43155                *.*                    LISTEN
Kliis:TCP-IPv4-v2 katjusha$ lsof -i TCP:43155
COMMAND     PID     USER   FD   TYPE             DEVICE SIZE/OFF NODE NAME
TCPServer 31019 katjusha    3u  IPv4 0xb95a848728ee5b81      0t0  TCP *:43155 (LISTEN)
TCPClient 31174 katjusha    3u  IPv4 0xb95a8487297672d1      0t0  TCP localhost:65195->localhost:43155 (ESTABLISHED)
TCPServer 31175 katjusha    4u  IPv4 0xb95a848727797b81      0t0  TCP localhost:43155->localhost:65195 (ESTABLISHED)
Kliis:TCP-IPv4-v2 katjusha$
```

netstat and **lsof** commands on the Mac OS X where the two applications were running

lsof for sockets

```
TCP-IPv4-v2 — bash — 118×19 — ⌘4
Kliis:TCP-IPv4-v2 katjusha$ netstat -a -p tcp | grep 43155
tcp4       0      0  localhost.43155        localhost.65195        ESTABLISHED
tcp4      52      0  localhost.65195        localhost.43155        ESTABLISHED
tcp4       0      0  *.43155                *.*                    LISTEN
Kliis:TCP-IPv4-v2 katjusha$ lsof -i TCP:43155
COMMAND     PID     USER   FD   TYPE             DEVICE SIZE/OFF NODE NAME
TCPServer 31019 katjusha    3u  IPv4 0xb95a848728ee5b81      0t0  TCP *:43155 (LISTEN)
TCPClient 31174 katjusha    3u  IPv4 0xb95a8487297672d1      0t0  TCP localhost:65195->localhost:43155 (ESTABLISHED)
TCPServer 31175 katjusha    4u  IPv4 0xb95a848727797b81      0t0  TCP localhost:43155->localhost:65195 (ESTABLISHED)
Kliis:TCP-IPv4-v2 katjusha$
```

nestat on Mac OS X

lsof -i protocol:port

lsof command on the Mac OS X where the two applications were running.
With respect to socket descriptors, the command outputs:

COMMAND: name of the UNIX command associated with the process
PID: process'pid
USER: user to whom the process belongs
FD: File Descriptor number of the file (**u** *means read and write access*)
TYPE: IPv4, for an IPv4 socket
NODE: Internet protocol type - e. g, "TCP"
NAME: the local and remote Internet addresses of a network file:
local host name or IP number, followed by a colon (:), the port, "->", and the two-part remote address

netstat

```
$ netstat -a -p tcp
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address      Foreign Address        (state)
tcp4       0      0  localhost.43155    localhost.65195        ESTABLISHED
tcp4       52     0  localhost.65195    localhost.43155        ESTABLISHED
tcp4       0      0  *.43155            *.*                    LISTEN
```

netstat command on the Mac OS X where the two applications were running:
-a: state of all sockets
-p: specify a protocol

Proto: the socket protocol
Recv-Q: number of bytes in the socket receive buffer, yet unread by local application (for UDP the field counts data + UDP headers + metadata)
Send-Q: number of bytes queued for transmission in the socket send buffer (for UDP sockets, the field counts data + UDP headers + metadata)
Local Address: {host IP + port} or {* + port} if the IP wildcard address is used
Foreign Address: {remote IP + port} or {*:*} if no remote peer address is set
state: current state of the socket

netstat

- netstat with the ***-r*** option shows the node's routing table

```
kat@hplinux2:~$ netstat -r
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
default         193.204.77.127  0.0.0.0         UG        0 0          0 eth0
193.204.77.0    *               255.255.255.0   U         0 0          0 eth0
kat@hplinux2:~$
```

- netstat ***-g*** shows information related to multicast group membership
- netstat options differ from Linux to OS X:
  - on Linux:
    - *-p* shows the pid of the process, *-protocol* to specify address family
      - *netstat -a --inet*
  - on Mac OS X:
    - *-p* specify the protocol family
      - *netstat -a -p tcp*

Socket creation

the communication domain, the type of service and the protocol to use

Catiuscia Melle and Franco Tommasi - SNP 2021-2022

Socket creation

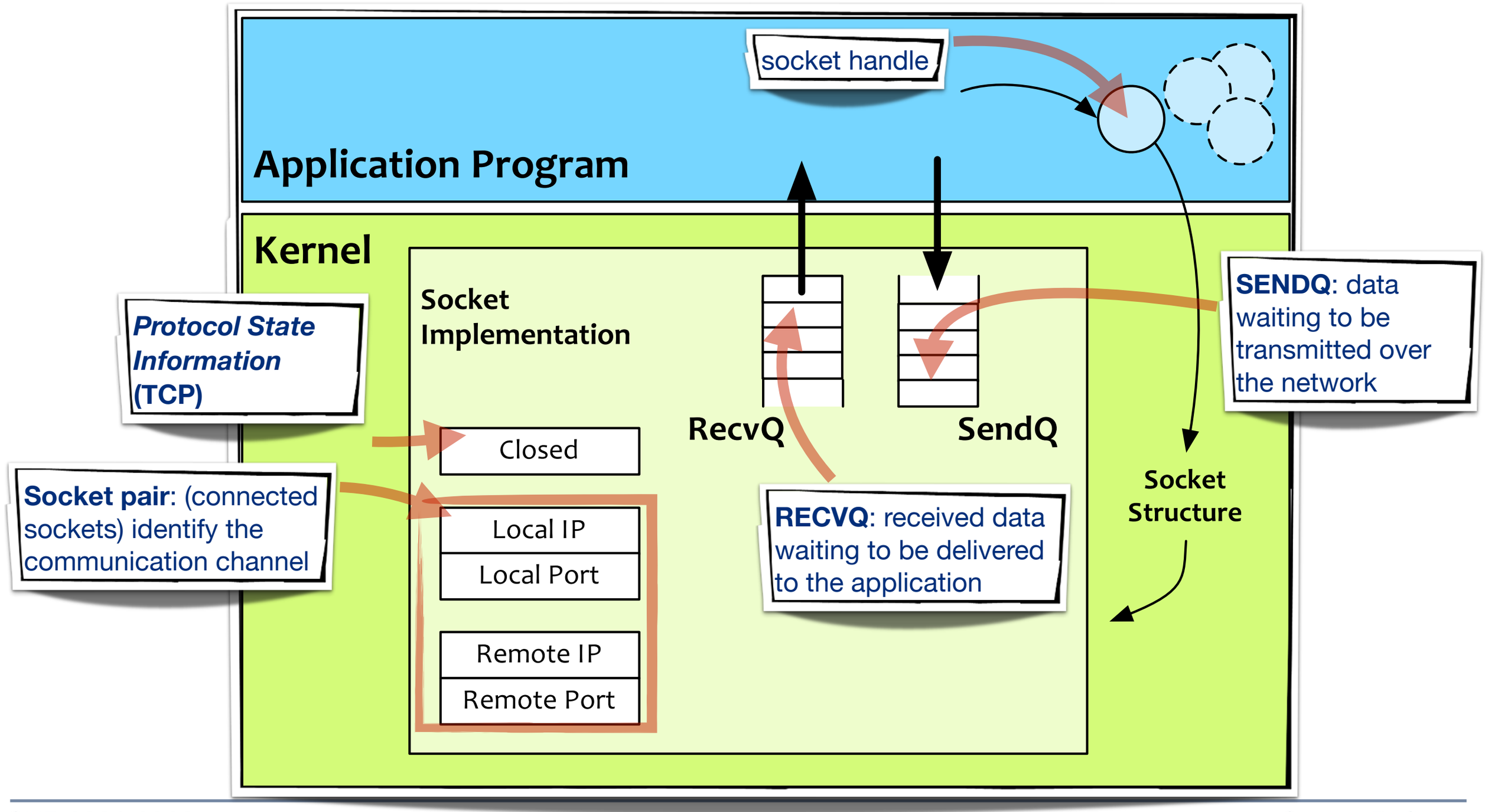
- The ***socket()*** system call creates a socket and returns the socket file descriptor:

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);

Returns the socket descriptor on success, -1 on error
```

- The function allocates in the kernel the resources needed to manage the communications among remote processes.
- The call's parameters specify:
  - the **domain** in which the communication will take place
  - the **type of service** the socket will provide to the application
  - the **communication protocol** to be used in the selected domain

Socket creation

- No address whatsoever has been specified at this point. The only result so far is the allocation of a structure in the kernel, which might look like this:



Socket creation parameters

- **domain**: is represented by an *Address Family*
  - **AF_UNSPEC**: Unspecified
  - **AF_INET**: Internet domain sockets for use with IPv4
  - **AF_INET6**: the IPv6 version
  - **AF_UNIX**: UNIX domain sockets
    - *IPC on the same host (by means of a socket type "file")*
      - a sort of bidirectional pipe
- historically, domain was represented by a *Protocol Family* (PF_XXX)
  - *PF_INET* was the **domain** in *socket()*, *AF_INET* was the **address family** in the *struct sockaddr_in*
  - in current implementations *PF_XXX* is defined in terms of *AF_XXX*
    - PF_XXX is not present in SuSv3 nor in SuSv4 - (#include <sys/socket.h>)

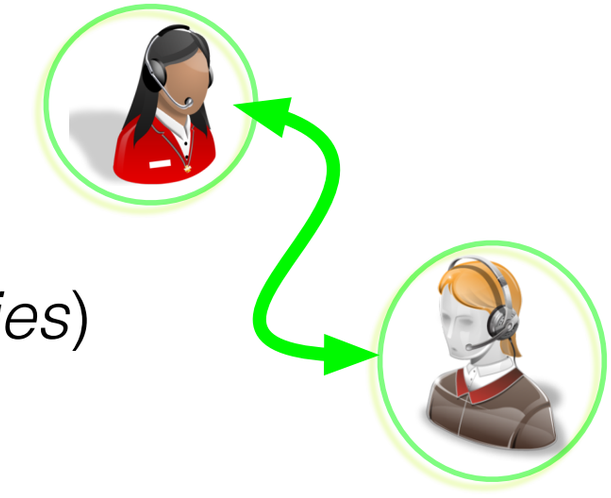
on Linux:
```
#define PF_INET 2
#define PF_INET6 10
#define AF_INET PF_INET
#define AF_INET6 PF_INET6
```

on MAC OS X:
```
#define AF_INET 2
#define AF_INET6 30
#define PF_INET AF_INET
#define PF_INET6 AF_INET6
```

Socket type: Datagram vs Stream

- **type**: service model (or *communication semantics*)
  - For AF_INET/AF_INET6 domains, it has one out of two values:
    - ***SOCK_DGRAM***
    - ***SOCK_STREAM***
- **DGRAM** socket
  - connectionless
  - unreliable
  - datagram oriented (*it keeps message boundaries*)

- **STREAM** socket
  - **connection oriented**:
    defined by a *socket pair* (*local socket*, *remote socket)*
  - **reliable**, **bidirectional**
  - **byte-stream** communication channel (*no message boundaries*)



Transport protocol

- **protocol**: which protocol to use with the socket (must be supported by the Address Family for the desired type of service)
  - ***0*** (default value) selects the default protocol for the chosen *type* and *domain*
- For the TCP/IP stack (AF_INET/AF_INET6 domains) the default protocol are:
  - ***protocol TCP (IPPROTO_TCP)*** for ***SOCK_STREAM*** type sockets
  - ***protocol UDP (IPPROTO_UDP)*** for ***SOCK_DGRAM*** type sockets
- The socket type choice reflects needs and constraints of the network application to be implemented. So does the chosen protocol.

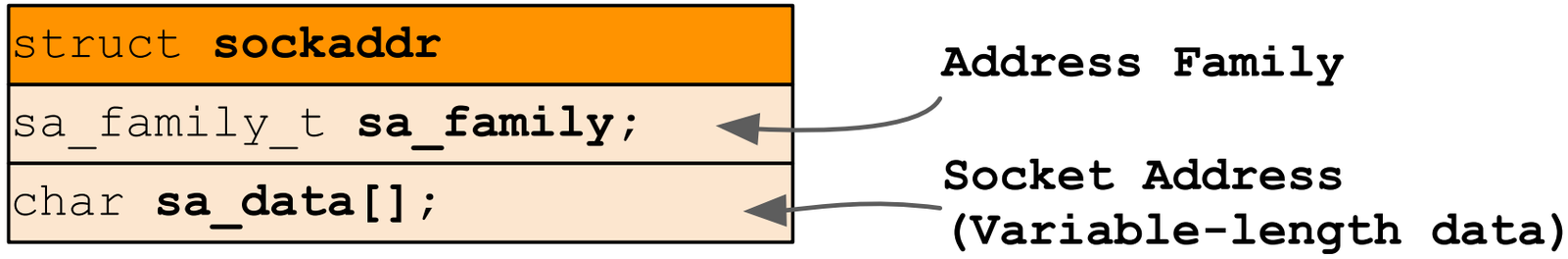
Socket addresses

Network addresses
Network byte order vs Host byte order
Presentation-to-numeric
Numeric-to-presentation

Catiuscia Melle and Franco Tommasi - SNP 2021-2022

Socket Addresses

- **<sys/socket.h>** defines the **struct sockaddr**, which must contain:



  - *sa_family* is the domain the struct will be used for (*AF_INET, AF_INET6, ...*)
  - *sa_data* is a generic type which will store the address for the selected Address Family.
    - Most implementations add a ***sa_len*** field (type *uint8_t*) contenente la dimensione della struttura (ed eventuali campi di *padding* per allineamento)

- Such an *opaque* structure, the API becomes ***generic*** and may be used in different domains
- Each socket domain supports a specific Address Family
- Each Address Family uses a different address format
  - *pathname* for AF_UNIX, *IPv4* for AF_INET, *IPv6* for AF_INET6

Socket addresses in domain INETx

- In TCP/IP networks, a *socket address*s is two pieces of information : *IP address + service port* (TCP/UDP)
  - the **IP address** identifies an *host* in the selected domain
  - the **service port** identifies the *process* - on the same host - whom data are intended for
    - the IP address is associated with *a network interface*
    - the service port identifies a socket in the system

- The complete address of a socket - aka ***Socket Pair*** is:

Local IP:local port	Remote IP: remote port	protocol (TCP/UDP)

IPv4 and IPv6 addresses

- An **IPv4** address consists of *4 byte*, aka *octets*, or 32 bits

  2^{32} => addresss space: **~4,3x10^{9}** IPv4 addresses

  - To make memorization and communication among humans easier, they are expressed in a *quad-dotted-decimal notation*
    - e.g., a binary IPv4 address:

      00001010 00000000 00001100 11110011

    - is expressed in quad-dotted-decimal notation as:

      10.0.12.243

IPv4 and IPv6 addresses

- An **IPv6** address consists of *128 bits*, (16 bytes)

  2^{128} => `addresss space:` **~3,4x10^{38}** `IPv6 addresses`

- they are expressed in the form (***full notation***):

  `hhhh:hhhh:hhhh:hhhh:hhhh:hhhh:hhhh:hhhh`

  - where h is an hexadecimal digit
    - Example:

      `E3D7:0000:0000:0000:51F4:9BC8:C0A8:6420`

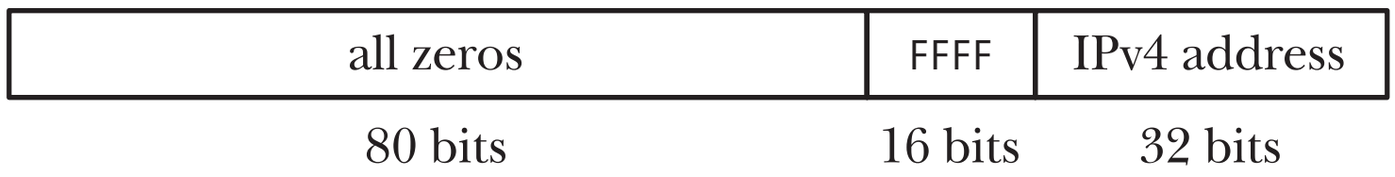
- the ***shorthand notation*** allows to drop from the IPv6 address the *leading zero*:
  - a sequence of null bytes may be substituted by "**::**"

    `E3D7::51F4:9BC8:C0A8:6420`

Dual stack IP implementation

- An operating systems tries to ease the IPv4-to-IPv6 transition by a technology called **dual-stack protocol**
  - Usually the OS implements both IP protocols (IPv4 ed IPv6), in an hybrid form (*RFC 4213*)
  - This choice allows programmers to write code capable of operating trasparently both on IPv4 and IPv6 networks, using hybrid sockets that may accept both types of packets

- To that end, IPv4 addresses are represented in the ***IPv4-mapped IPv6*** format



- Usually a ***IPv4-mapped IPv6*** address is represented by the first 96 bit in the standard IPv6 format and the remaining 32 bit in IPv4 quad-dotted-decimal notation.
  - Example: the IPv6 address **::ffff:192.0.2.128** corresponds to the IPv4 address 192.0.2.128
  - The ***IPv4-compatible IPv6*** *format* which used to set to **0** the first 96 bit, as in ***::192.0.2.128*** is now deprecated

Transport level addresses

- The TCP/UDP address is encoded in a *16 bit* (2 byte) field
  - 2^16 => 65536 porte (both for UDP and TCP)
- The assignement of port numbers to each Internet è managed by **IANA**

  http://www.iana.org
- Such assignments may be found in the file

  /etc/services
- The 65536 ports are partitioned in ranges:
  - *well-known*
  - *registered*
  - *ephemeral*

Transport level addresses

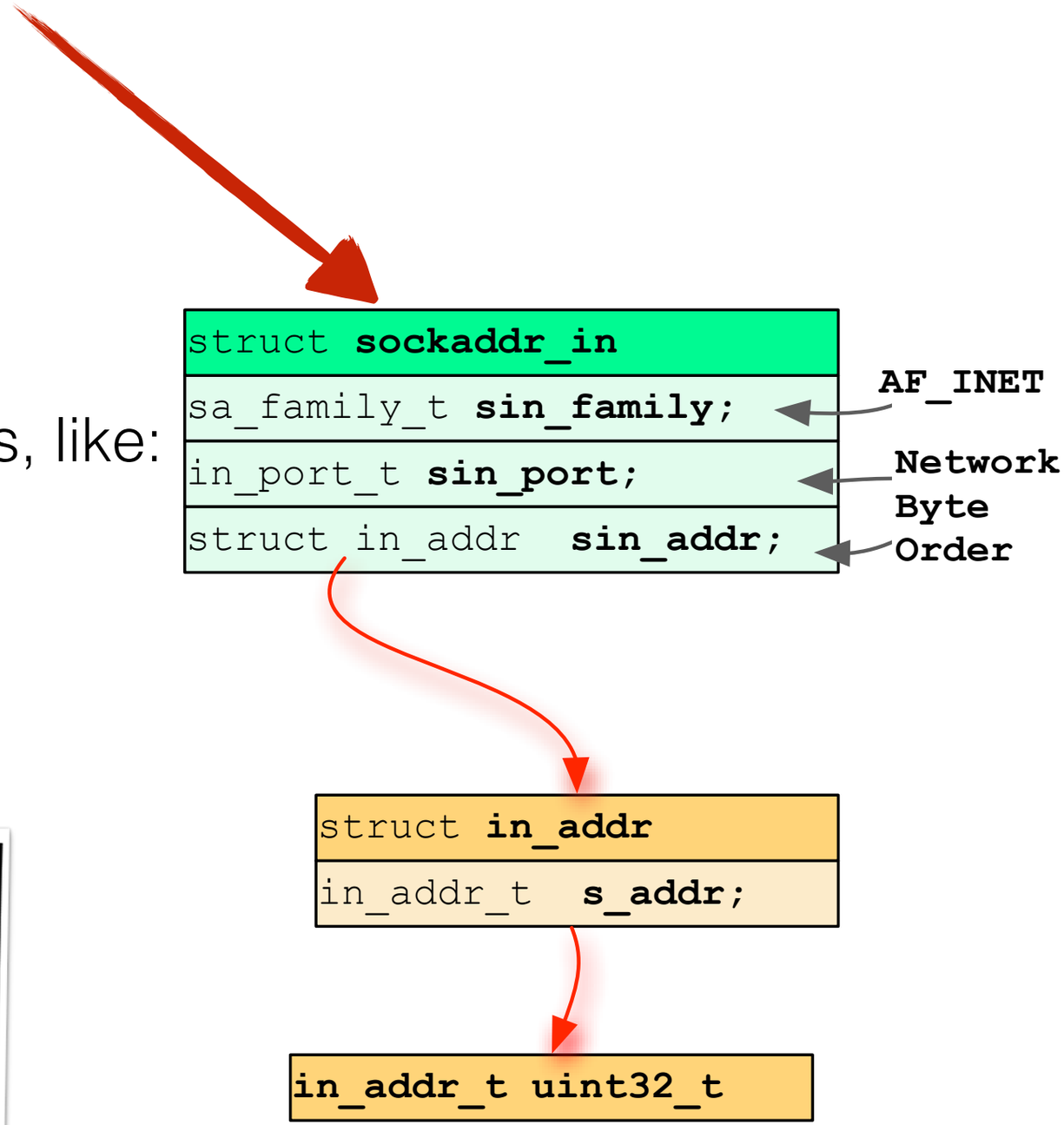
- **Well-known**:
  - ports from **0** to **1023**
  - controlled and assigned by IANA
  - In UNIX they are *reserved:* a process must own **superuser** privileges to use them.
- **Registered**:
  - ports from **1024** to **49151**
  - registered (and **not** controlled) by IANA.
- **Ephemeral or private or dynamic**:
  - ports from **49152** to **65535** (used to start connections from them to servers)

IPv4 and IPv6 sockets addresses

- To be able to represent IPv4 and IPv6 sockets addresses a new data type is needed (it cannot be ***struct sockaddr***, the one specified in Socket APIs)
- Types defined in **<netinet/in.h>** are used:
  - **struct sockaddr_in** for IPv4
  - **struct sockaddr_in6** for IPv6
- Since Socket APIs refer exclusively to ***struct sockaddr***, *struct sockaddr_in* and *struct sockaddr_in6* must be always **casted** to the generic type, in *all* calls.
- Beside, such structures must always be passed by *reference* to the APIs and in some calls they are *value-result argument:*
  - those which the kernel returns to the process, on call's completion

IPv4 sockets addresses

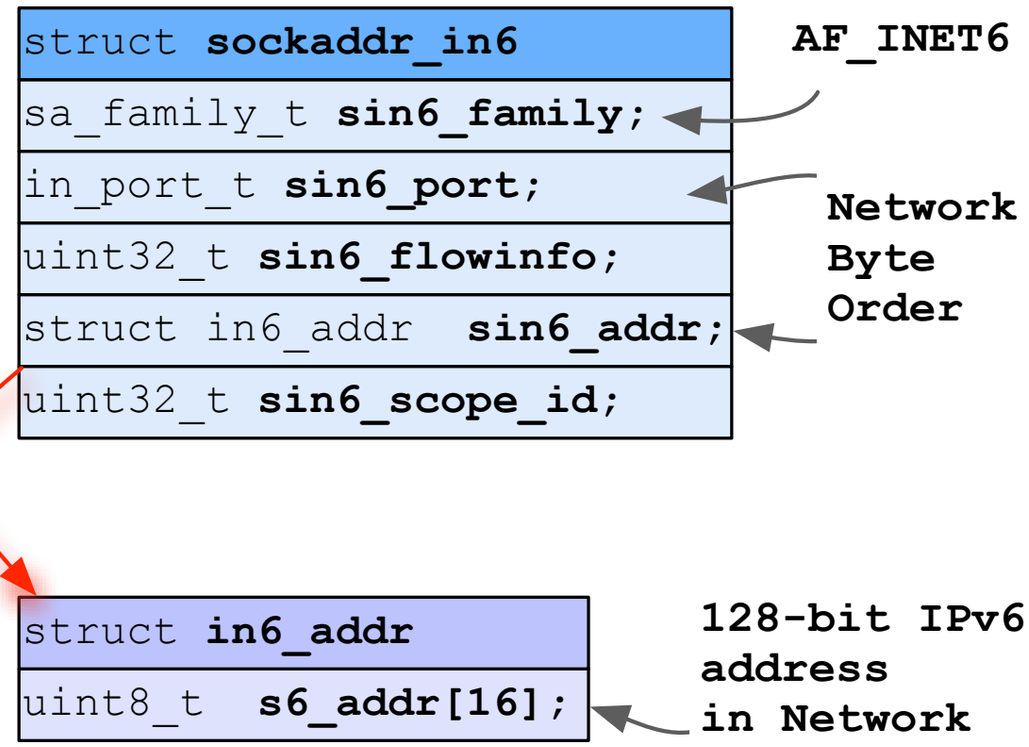
- SuSv3 and SuSv4, in *<netinet/in.h>*, define the address structure for the IPv4 domain:
  - address family (**AF_INET**)
  - IPv4 address, in *Network Byte Order*
  - TCP/UDP port, in *Network Byte Order*
- Some implementations add more fields, like:
  - *sin_len*, structure's length
  - *sin_zero*, zero-padding to 16 bytes




```
Su MAC OS X:
struct sockaddr_in {
 __uint8_t       sin_len;
 sa_family_t     sin_family;
 in_port_t       sin_port;
 struct in_addr  sin_addr;
 char            sin_zero[8];
};
```

IPv6 sockets addresses

- Likewise, **struct sockaddr_in6** specifies:
  - address family (**AF_INET6**)
  - IPv6 address, in *Network Byte Order*
  - TCP/UDP port, in *Network Byte Order*



- The MacOS implementation includes a *sin6_len* field

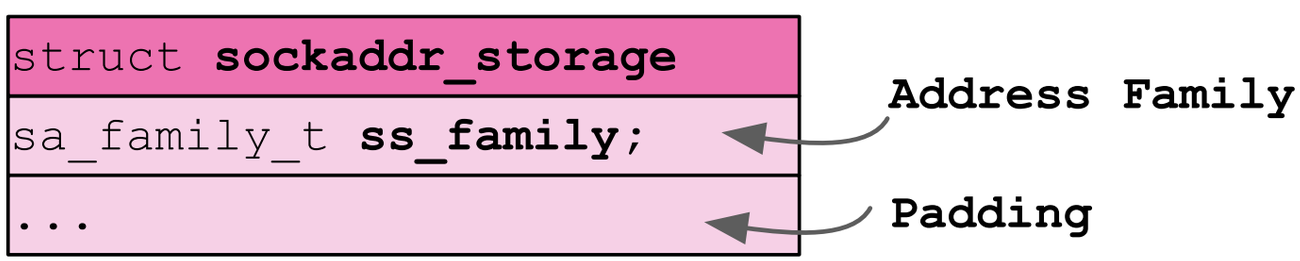
```
On MacOS in netinet6/in6.h:
struct sockaddr_in6 {
      __uint8_t sin6_len;   // length of this struct(sa_family_t)
      sa_family_t    sin6_family; // AF_INET6
      in_port_t sin6_port;      //Transport layer port #
      __uint32_t     sin6_flowinfo; // IP6 flow information
      struct in6_addr     sin6_addr;     // IP6 address
      __uint32_t     sin6_scope_id; // scope zone index
};
```

```
On Linux, in /usr/include/linux/in6.h:
struct sockaddr_in6 {
      unsigned short int sin6_family; //AF_INET6
      __be16 sin6_port;       // Transport layer port #
      __be32 sin6_flowinfo;  // IPv6 flow information
      struct in6_addr sin6_addr;  // IPv6 address
      __u32 sin6_scope_id;  // scope id (new in RFC2553)
};
```

IPv4-IPv6 code portability

- To simplify the transition between the two versions of the protocol and to encourage the writing of *protocol-independent* code, the type **struct sockaddr_storage** (in *<sys/socket.h>*) was defined

- It is large enough to contain both IPv4 and IPv6 addresses and to deal with them transparently:

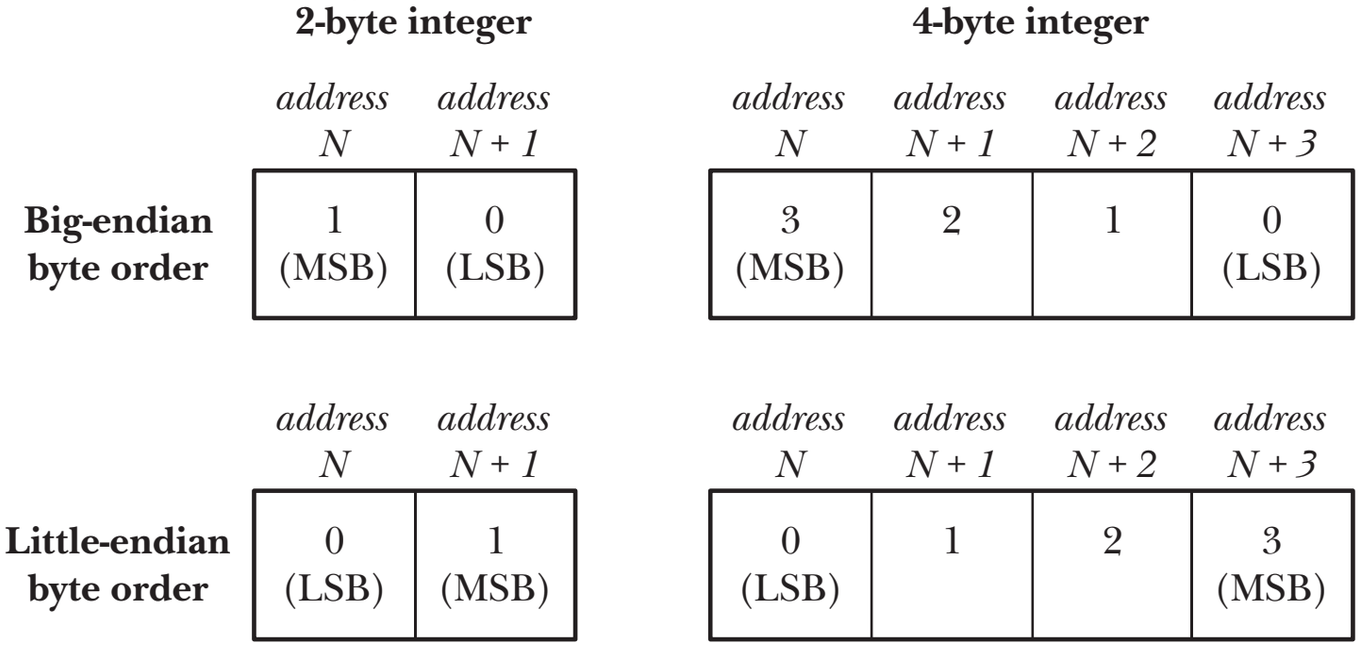


- The structure may handle all the Address Families thanks to the **ss_family** field

- To read the data it contains, the appropriate cast must be made:
  - *ss_family==AF_INET* -> cast to **struct sockaddr_in**
  - *ss_family==AF_INET6* -> cast to **struct sockaddr_in6**

Big Endian and Little Endian

- Each architecture has a different way to represent data types of **more than 1 byte** (no differences for 1 byte)

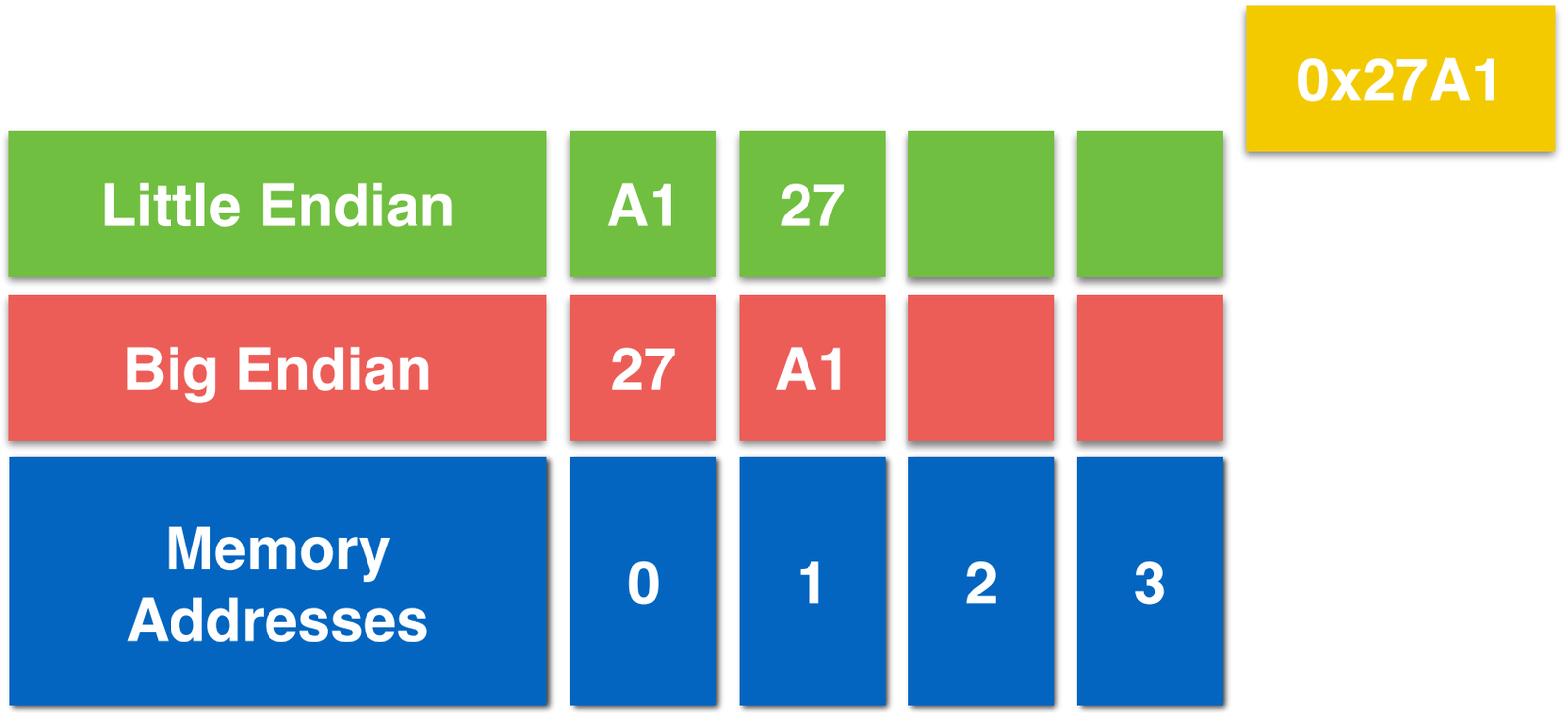


MSB = Most Significant Byte, LSB = Least Significant Byte

- The order by which data are read by from memory is called *host byte order* and, depending on what is memorized at the lowest address, it is called:
  - **Big Endian**, when the first byte is the *most* significant
  - **Little Endian**, when the first byte is the *least* significant
- While there is no consistency problem within each architecture, a rule must be agreed upon for exchanging data among architectures

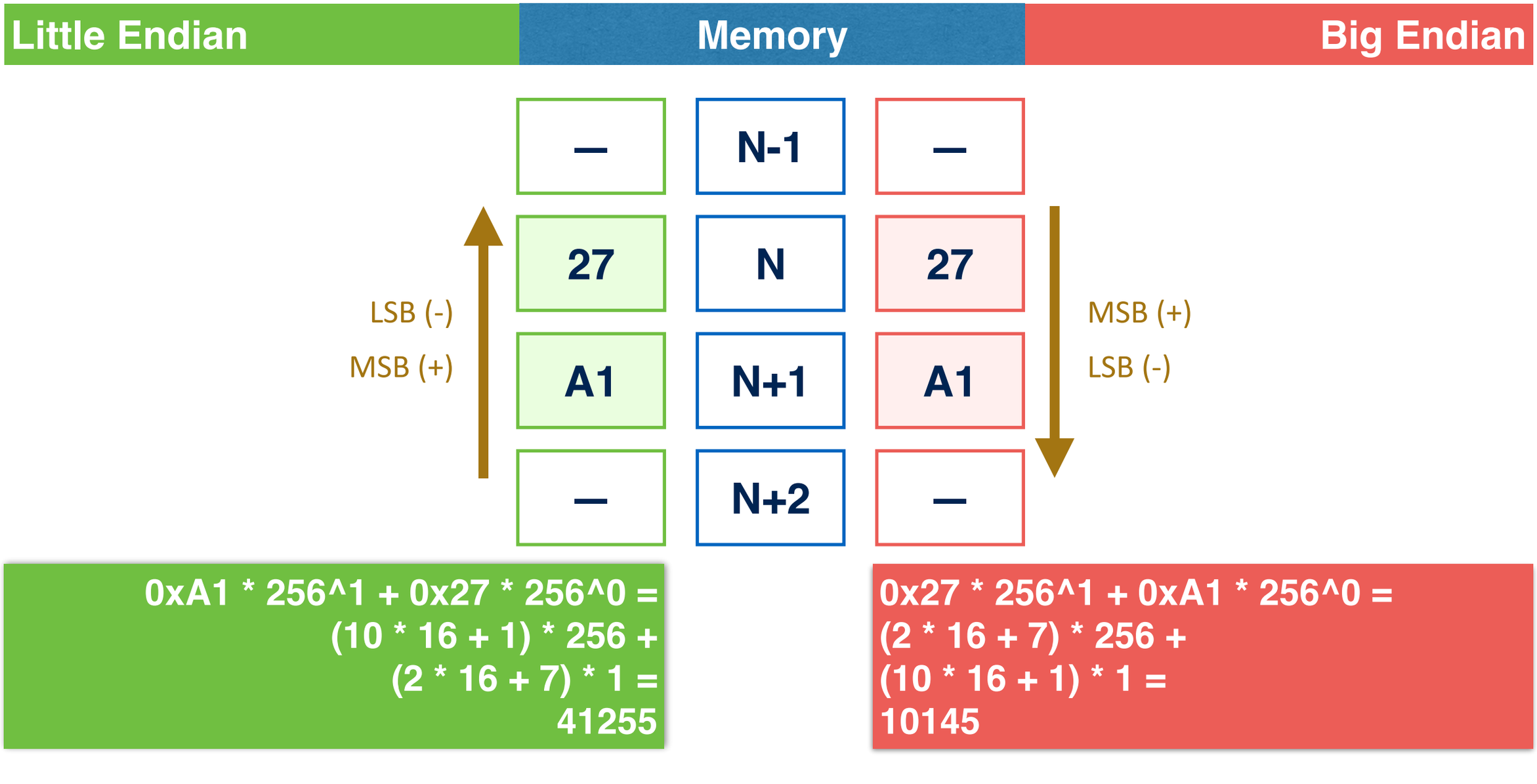
Big Endian vs Little Endian

- Let us consider the decimal value *10145*, HEX *0x27A1*
- It takes 2 bytes to represent it
- The picture shows how Little Endian and Big Endian architectures represent the data



Big Endian vs Little Endian

- If a multibyte sequence is exchanged among architectures with a different endianess, how must the bytes be interpreted?



Network Byte Order vs Host Byte Order

- All protocols encode their data in **Network Byte Order**
- **Network Byte Order** is the common exchange format and it coincides with the Big Endian Order.
- Before exchanging data among hosts, they must be converted to **Network Byte Order**. Then the can be transmitted on the network.
  - when received, the data must be converted to **Host Byte Order**. Then they can be processed.

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);

These functions shall convert 16-bit and 32-bit
quantities between
network byte order and host byte order.
```

IP addresses conversion

- To convert an IP adderss from the *numeric* form (binary value) to the textual form (string) and vice versa 2 functions may be used:

```
#include <arpa/inet.h>
const char *inet_ntop(int af, const void *restrict src,
                      char *restrict dst, socklen_t size);

int inet_pton(int af, const char *restrict src, void *restrict dst);
```

- **numeric to presentation** and **presentation to numeric** (or *network*)

- *af* represents the address family
  - it may be AF_INET or AF_INET6

- Both functions return -1 with error **EAFNOSUPPORT** when an unknown Address Family is provided (other than AF_INET or AF_INET6).
    - They substitute *inet_aton*, *inet_ntoa*, *inet_addr* (only IPv4, deprecated by SuSv4)

IP addresses conversion

- **inet_ntop()** returns:
  - **NULL** in case of error
  - or **a pointer to the buffer** containing the textual representation of the address pointed by *src* (in *network byte order*)
- The length of the strings where the conversion is stored may be defined using the following constants, defined in *<netinet/in.h>*:

```
#define INET_ADDRSTRLEN   16 //IPv4 dotted-decimal
#define INET6_ADDRSTRLEN  46 //IPv6 hex string
```

- **inet_pton()** returns:
  - **1** if the conversion succeeds
    - *dst* will receive the numeric IP address in network bye order
  - **0** if the input string is not a valid representation of an IP address
    - IPv4/IPv6 non parseables, according to the specified address family
  - **-1** in case of error

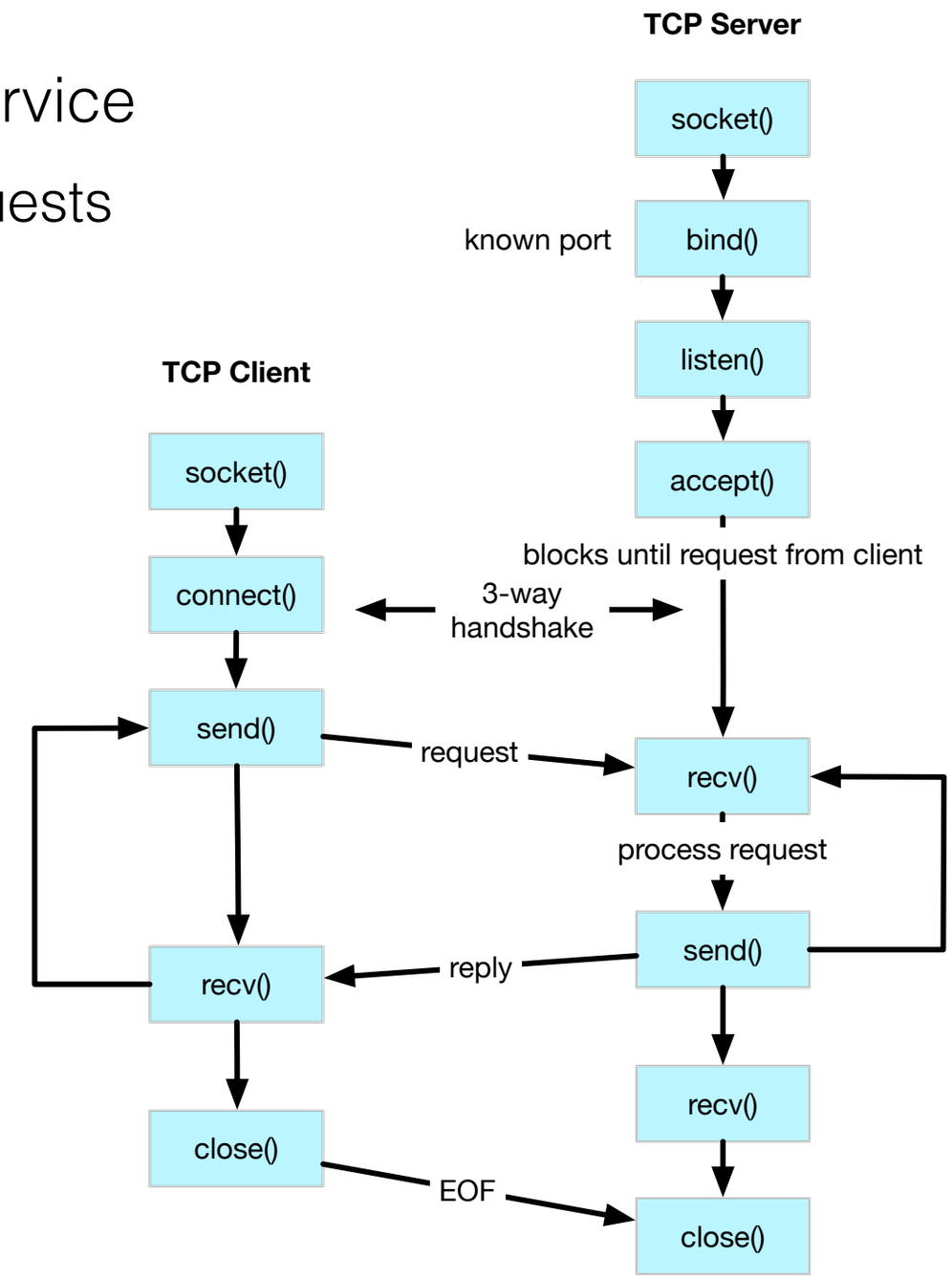
TCP client e TCP server

TCP workflow

Catiuscia Melle and Franco Tommasi - SNP 2021-2022

TCP client and TCP server

- A TCP-based application has to implement the *client-server* model.
  - The *server* has to get ready for the service
    - it must be able to receive clients' requests
  - The *client* starts the interaction
  - by contacting the server
- The workflow is shown in the picture
  - First, the server process, performs a **passive open** (*socket, bind, listen*)
  - After that, the clients may contact it (on a well-known port), carrying out an **active open** (*connect*)



Server: the service address must be specified

- After its creation, a socket *may* be associated to an address
  - *protocol address* must be appropriate for the communication domain
- To be reached by its clients, a *server* must be listening at a *well-known* address
  - The well-known address consists of an *IP* address and a *service port*

```
#include <sys/socket.h>
int bind(int socket, const struct sockaddr *address,
         socklen_t address_len);

Returns 0 on success, -1 on error.
The <sys/socket.h> header shall define the socklen_t type,
which is an integer type of width of at least 32 bits.
```

- **address** is a pointer to the address structure (*local*) we wish to associate with the socket (by a cast to *struct sockaddr*)
- The address format depends on the considered domain (AF_INET, AF_INET6)
- **address_len** is the size of the sockaddr actually used (often given with *sizeof*())

Server: the service address must be specified

- *bind()* is usually called by a server process (TCP/UDP)
- a client has no need to assign an address to its socket
  - the kernel will set the local address of the socket
  - such address will consists of a (free) *ephemeral port* and an IP address (corresponding to the network interface by which the traffic is forwarded to the server — depending on the node's routing table)
- By the *bind()* call a server chooses both the service port (must be well-known lest the client wouldn't know it) and the IP address (a specific one or the null *wildcard address)*:
  - IPv4:
    sin_addr.s_addr = htonl(**INADDR_ANY**)
  - IPv6:
    sin6_addr = **in6addr_any**

Server: the service address must be specified

- The standard in *<netinet/in.h>* defines the *IP wildcard addresses*:
  - **IPv4**:

```
#define INADDR_ANY (u_int32_t)0x00000000
```

  - **IPv6**:

```
Su MAC OS X in <netinet6/in6.h>:
extern const struct in6_addr in6addr_any;

Su Linux in <netinet/in.h>
extern const struct in6_addr in6addr_any;
```

    - The **in6addr_any** struct is set by the system with the **IN6ADDR_ANY_INIT** *macro* which defines an array of 0 [array can be initialized (but not assigned) at once]:

```
#define IN6ADDR_ANY_INIT {{{ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 }}}
```

- In *<netinet/in.h>* also the address corresponding to the *loopback interface* can be found [array can be initialized (but not assigned) at once]:
  - **IPv4**: **INADDR_LOOPBACK** *(127.0.0.1)*
  - **IPv6**: *const struct in6_addr* **in6addr_loopback** *(::1)*
    - **Beware**: the IPv6 wildcards **are** in Network Byte Order but the IPv4 ones **are not** and the function **htonl()** must be used

Server: service set-up

- To accept TCP connection from a client, a server must be put in a *listen state:* its socket must be marked as *passive (**passive open**)*:

```
#include <sys/socket.h>
int listen(int socket, int backlog);

Returns 0 on success, -1 on error.
```

- The *backlog* parameter represents the size of the server's *pending connections queue*
  - Backlog is the sum of *pending connections* and *completed connections* the kernel is able to manage for a socket
    - that is those in the **SYN_RCVD** state and those already **ESTABLISHED** which completed the 3WHS
- First implementations had an backlog upper limit of 5.
  The SuSv3/SuSv4 standard defines the **SOMAXCONN** constant as *the maximum backlog queue length*
- In Linux e Mac OS X **SOMAXCONN** is **128**

Server: service set-up

- The server process will be able to manage the connections with the clients, collecting from the kernel all needed information:

```
#include <sys/socket.h>

int accept(int listening_socket,
           struct sockaddr *restrict address,
           socklen_t *restrict address_len);

Upon successful completion, accept() shall return the non-
negative file descriptor of the accepted socket. Otherwise, -1
shall be returned and errno set to indicate the error.
```

- *accept()* returns a ***new*** socket descriptor
  - The new socket descriptor identifies the connected socket pair (*Server IP:port, Client IP:port*) created after 3WHS completes
  - The **listening socket** remains listening for new requests
    - *address* and *address_len* are returned by the kernel to the process as *value-result arguments* contain the remote client protocol address and the structure size

Client: talk to server

- In client-server model, a client starts a new communication toward the server, whose *IP address and TCP service port* it knows.
- After the socket is created, the connection to the server is accomplished by the call:

```
#include <sys/socket.h>
int connect(int socket, const struct sockaddr *address,
                                        socklen_t address_len);
Returns 0 on success, -1 on error.
```

  - ***address*** is the ptr to the appropriate address structure for the considered domain - It is passed to the kernel by *reference* after a **cast** to *struct sockaddr.* It contains the address of the remote process (*server*) with whom the client wishes to establish an end-to-end connection.
  - For a TCP socket, *connect()* starts the 3WHS
- If *connect()* fails, SuSv3 specifies the *portabile* way to retry the connection is:
  - close the socket and open a new one
  - retry the connection with the new socket

Client: talk to server

- The client has no need to call *bind()* to assign an address to its socket
  - the kernel will set the local address of the socket assigning to it a (free) ephemeral port and an IP address (based on the node's routing table content)
- The client TCP layer sends a ***SYN*** to the remote server TCP layer:
  - If no answer is received, the kernel re-sends the SYN and starts a timer with an exponentially increasing delay
    - If at timeout (~75s) no answer has been received, the **ETIMEDOUT** error is returned
  - if no server process is available at the specified address a **RST** will be received
    - *hard error*: the kernel will return the **ECONNREFUSED** error to the application
  - If the kernel receives *ICMP* messages caused by network problems (*host unreachable, net unreachable*), it will continue sending SYNs until the timeout expiry
    - After timeout the kernel reports an **EHOSTUNREACH**/ **ENETUNREACH** error to the application (*soft error*)

Send and Receive

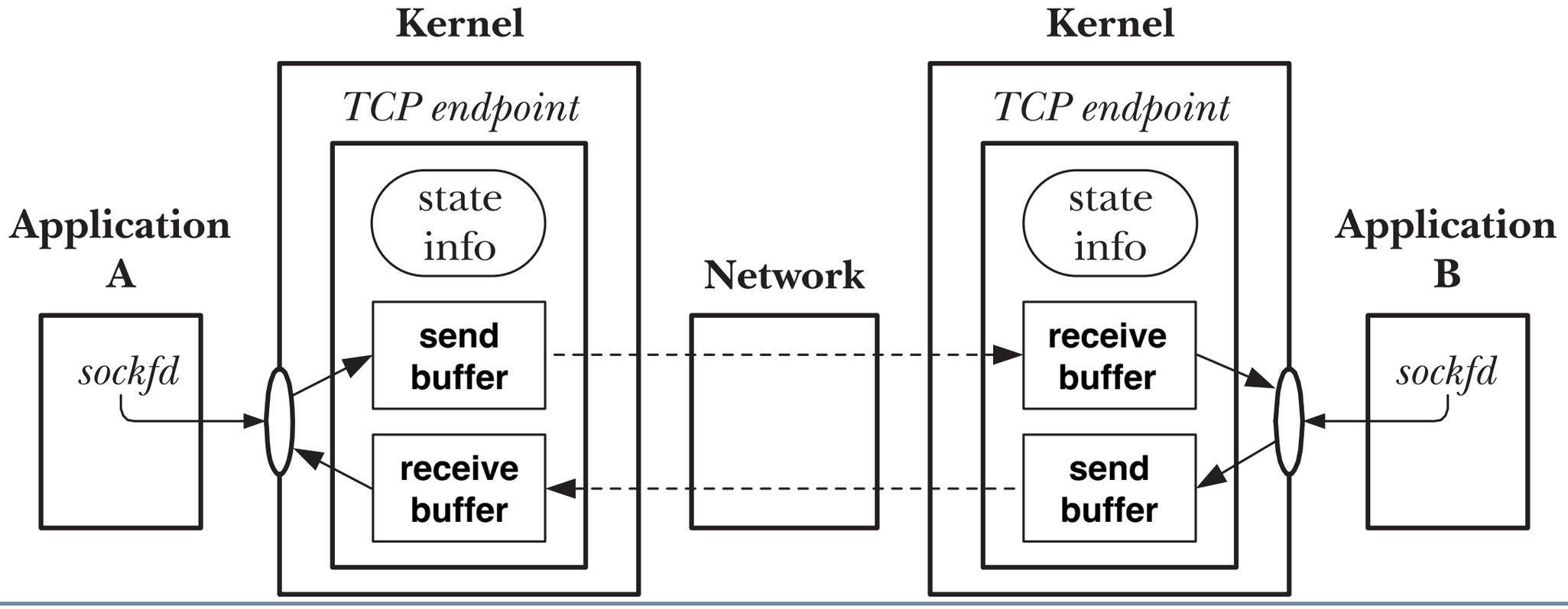
- When the connection is established the data exchange may begin
  - The two sockets are in the **ESTABLISHED** state
- Writing and reading from a socket is implemented by:

```
#include <sys/socket.h>

ssize_t send(int socket,const void *buf,size_t len,int flags);

ssize_t recv(int socket,void *buf,size_t len,int flags);
```

- **send()**: returns the number of bytes *queued* in the send queue
  - after the call returns, we don't know whether the data have been transmitted
- **recv()**: returns the number of bytes taken from the receive buffer
  - **0** if all the data from the receive buffer have been collected and the other end of the connection has been closed (*End-of-File*)

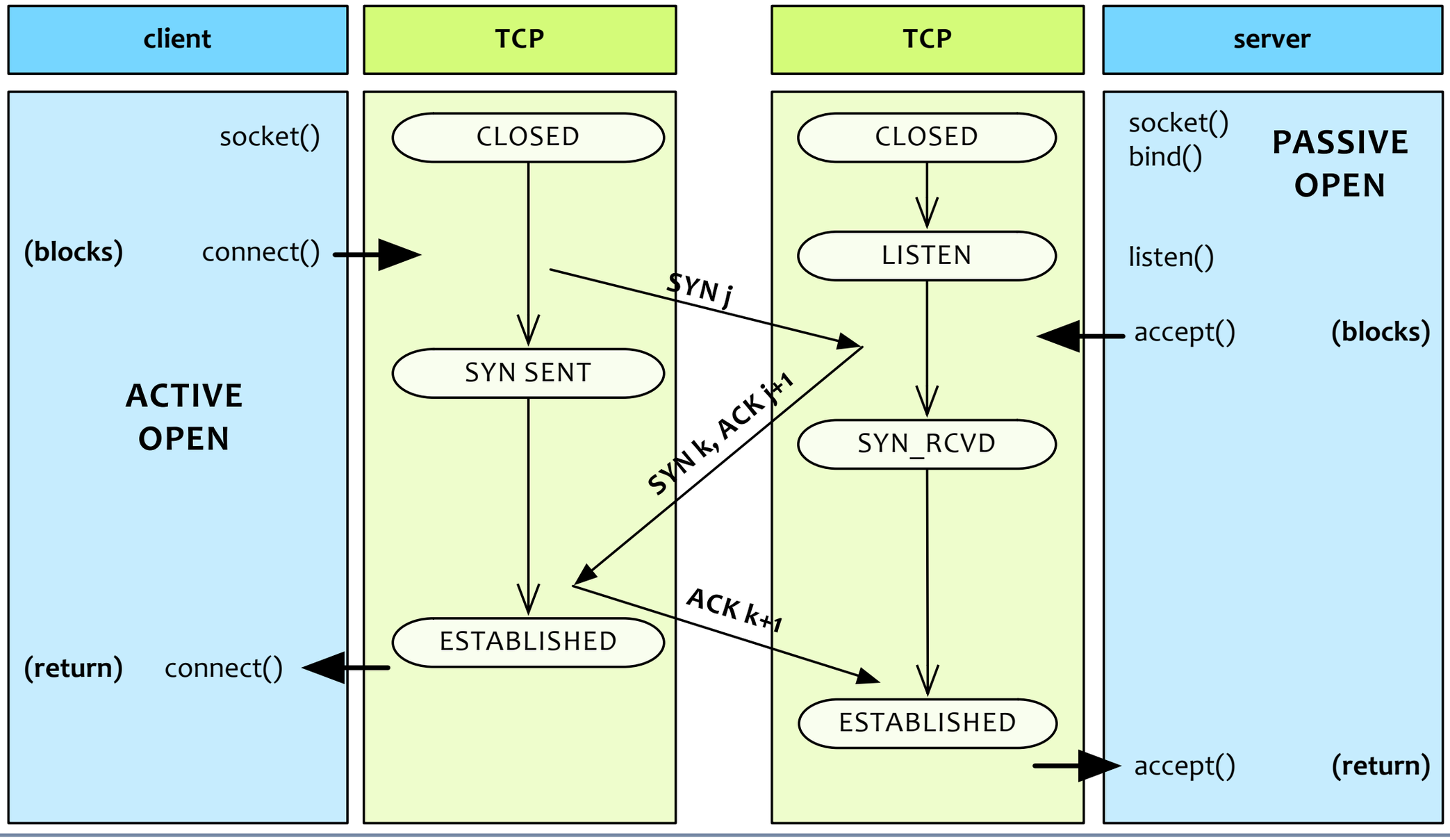
Send and Receive

- TCP data are split in 1 or more *segments*, each encapsulated in a datagram IP
- a *TCP segment* which is correctly received, is acknowledged by an *ACK* to the sender (cumulative ACK).
- When the ACK is received, the sender stops the retransmission timer for the segment — if not starts retransmitting
    - **RTO** is the *retransmission timeout* value, adapted to the network conditions
- **Flow Control**: together with each ACK the receiver sends information about the space available in the *receive buffer**, (*sliding window algorithm*) — [* option **SO_RCVBUF**]



TCP 3-way handshake

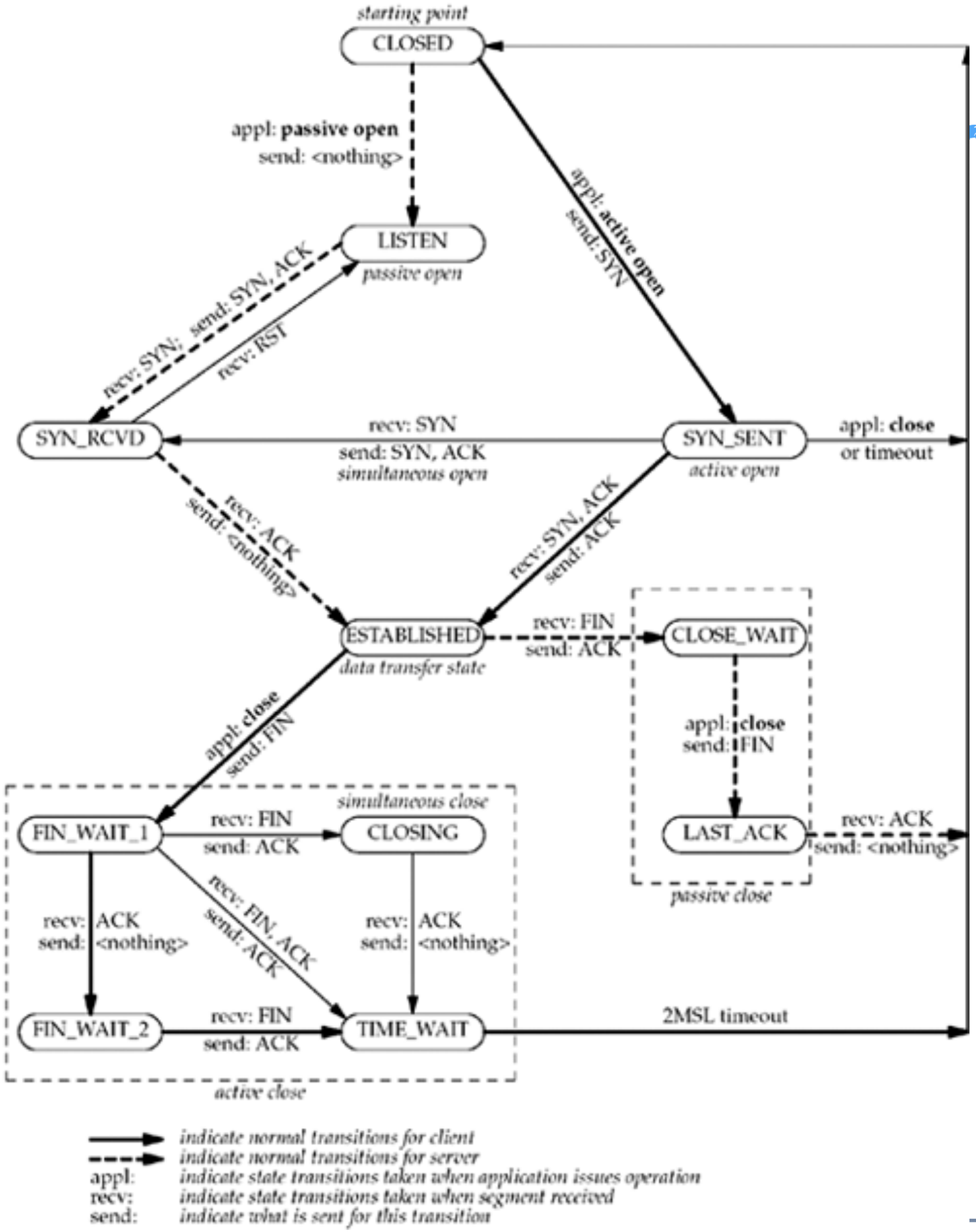
- The picture gives a summary of the steps needed to establish a TCP connection and the protocol state transitions
  - TCP options are exchanged during the 3WHS



TCP state diagram

from

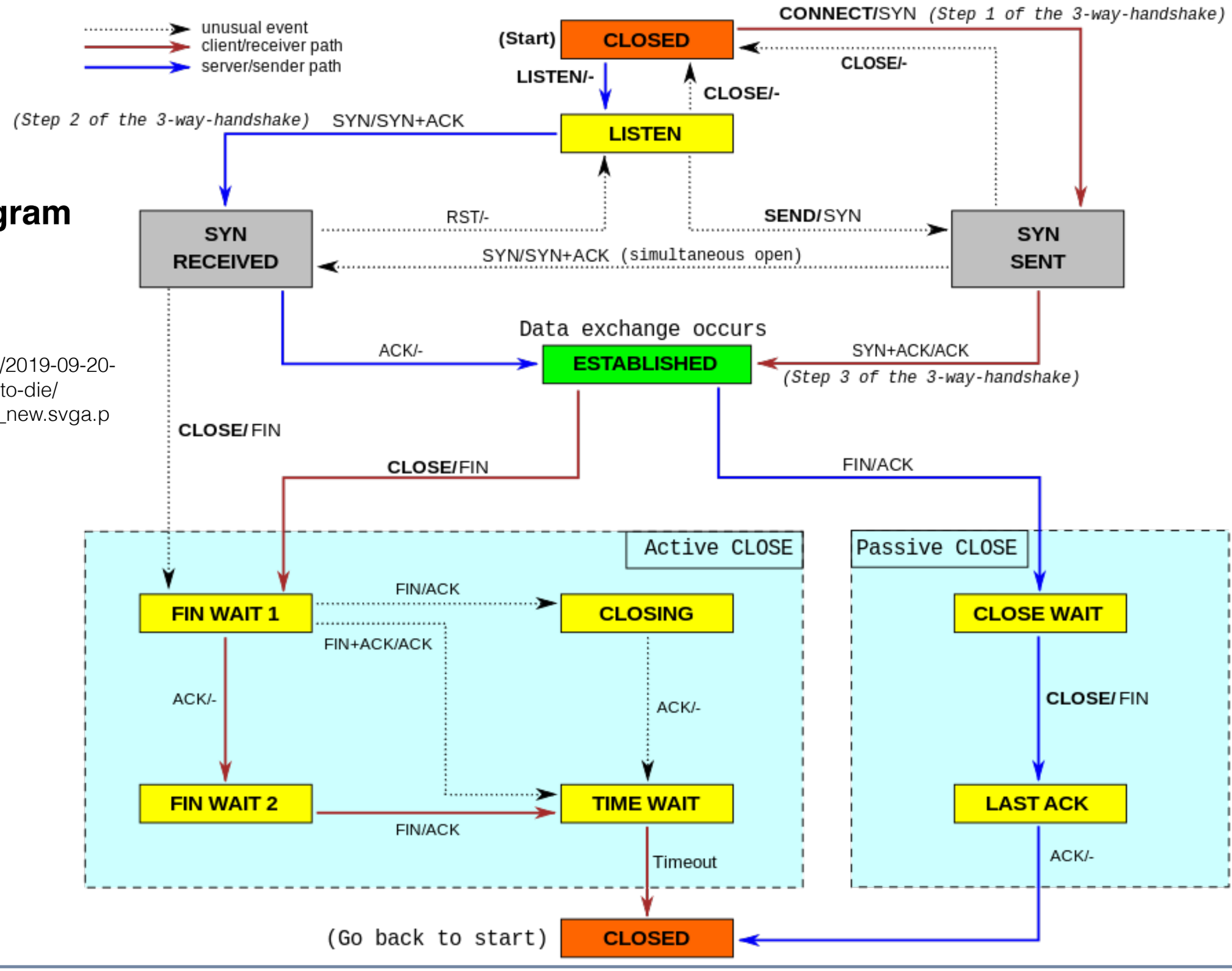
Stevens et al.
Unix Network Programming V.1 3rd ed.



TCP state diagram

from

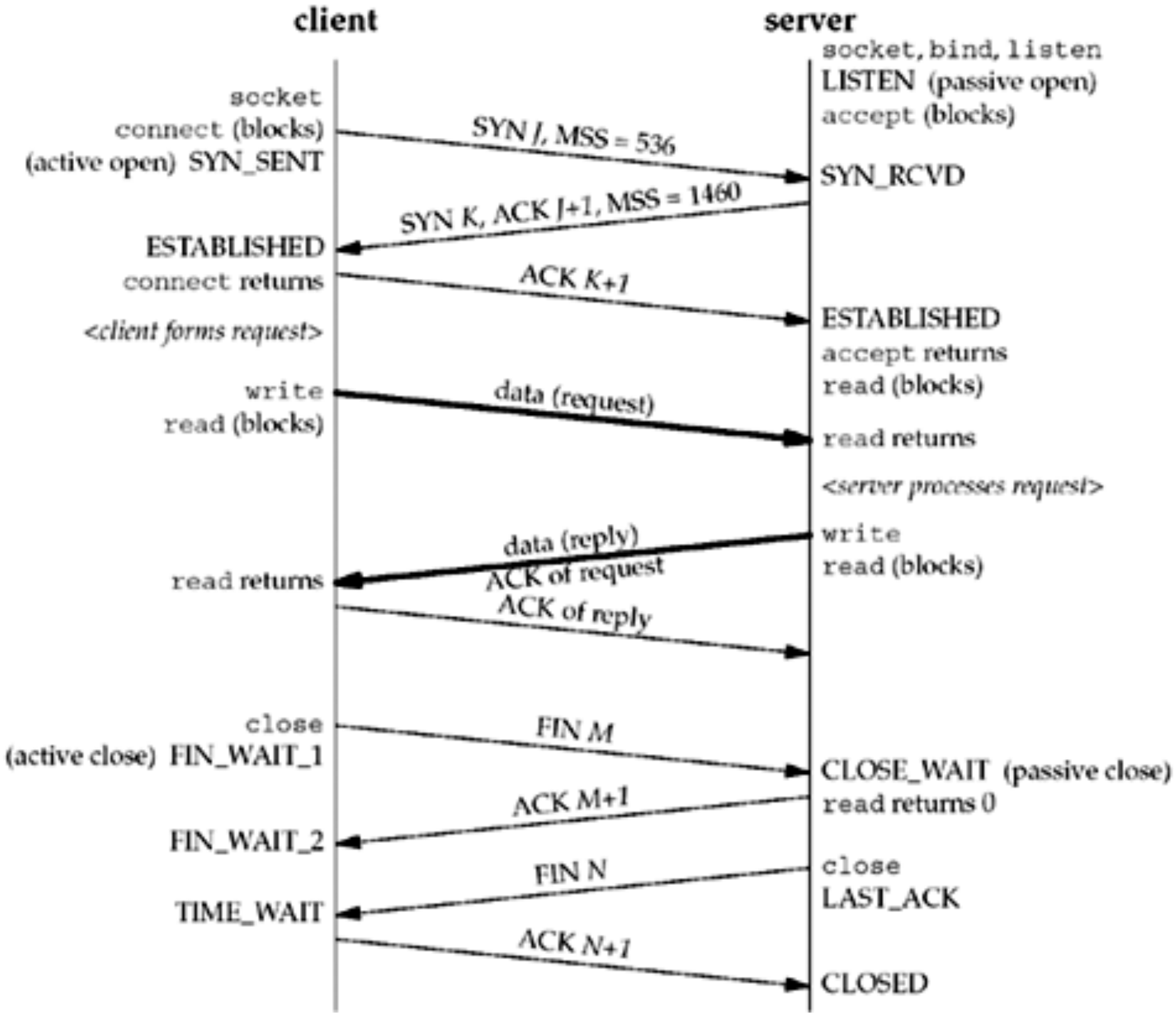
https://idea.popcount.org/2019-09-20-when-tcp-sockets-refuse-to-die/
Tcp_state_diagram_fixed_new.svga.png



Packet exchange for TCP connection

from

Stevens et al.
Unix Network Programming V.1 3rd ed.



Closing a connection

- Closing a socket terminates the communication in *both* directions

```
#include <unistd.h>

int close(int fd);
Returns 0 on success, -1 on error.
```

- For a TCP socket, the *close()* call marks a socket as closed (that is, no more usable by the process)
  - The TCP layer sends all the data still in the Send Queue before initiating the closing process (*4-Way teardown*)
  - The first process invoking *close()* puts into effect the *active close* of the TCP connection
- At any given moment, an application may hold **more than one handle for a socket**, that is more socket descriptor referencing it:
  - At each invocation, close() *decrements* the socket reference count
  - A socket is *in effect* when **reference count becomes 0**
    - Since *close()* closes both direction of the data transfer, both *reading and writing*, in some case data may remain not transferred between the two sockets

Local and Remote Address

- The **local** address of a connected socket is read by

```
#include <sys/socket.h>
int getsockname(int socket, struct sockaddr *restrict
                local_address,socklen_t *restrict address_len);
```

  - Set the passed structure with the local IP and the local port assigned to the socket
    - if *bind()*, has not been performed, after a *connect()*, to know IP address and port number assigned by the kernel
    - it works also for a connected UDP socket

- The address of the connected **remote node** whose handle the socket represents, can be read by the function:

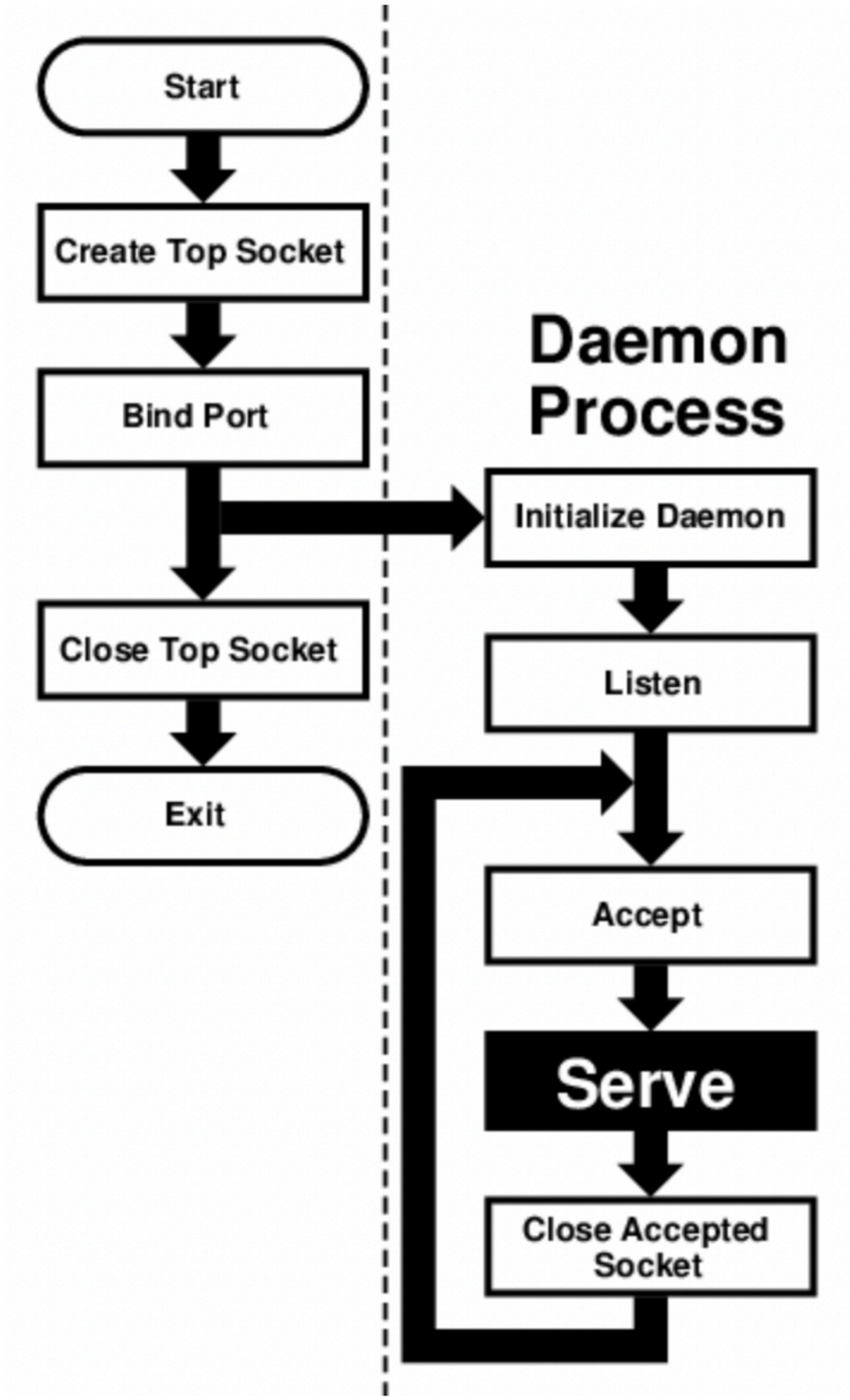
```
#include <sys/socket.h>
int getpeername(int socket, struct sockaddr *restrict
                remote_address,socklen_t *restrict address_len);
```

  - sets *remote_address* and returns its size in *address_len*
    - if the server invoked *accept()* specifying *NULL* for address and size
    - if a *child process* inherited a connected socket from the *parent*

Sequential (or iterative) Server

from

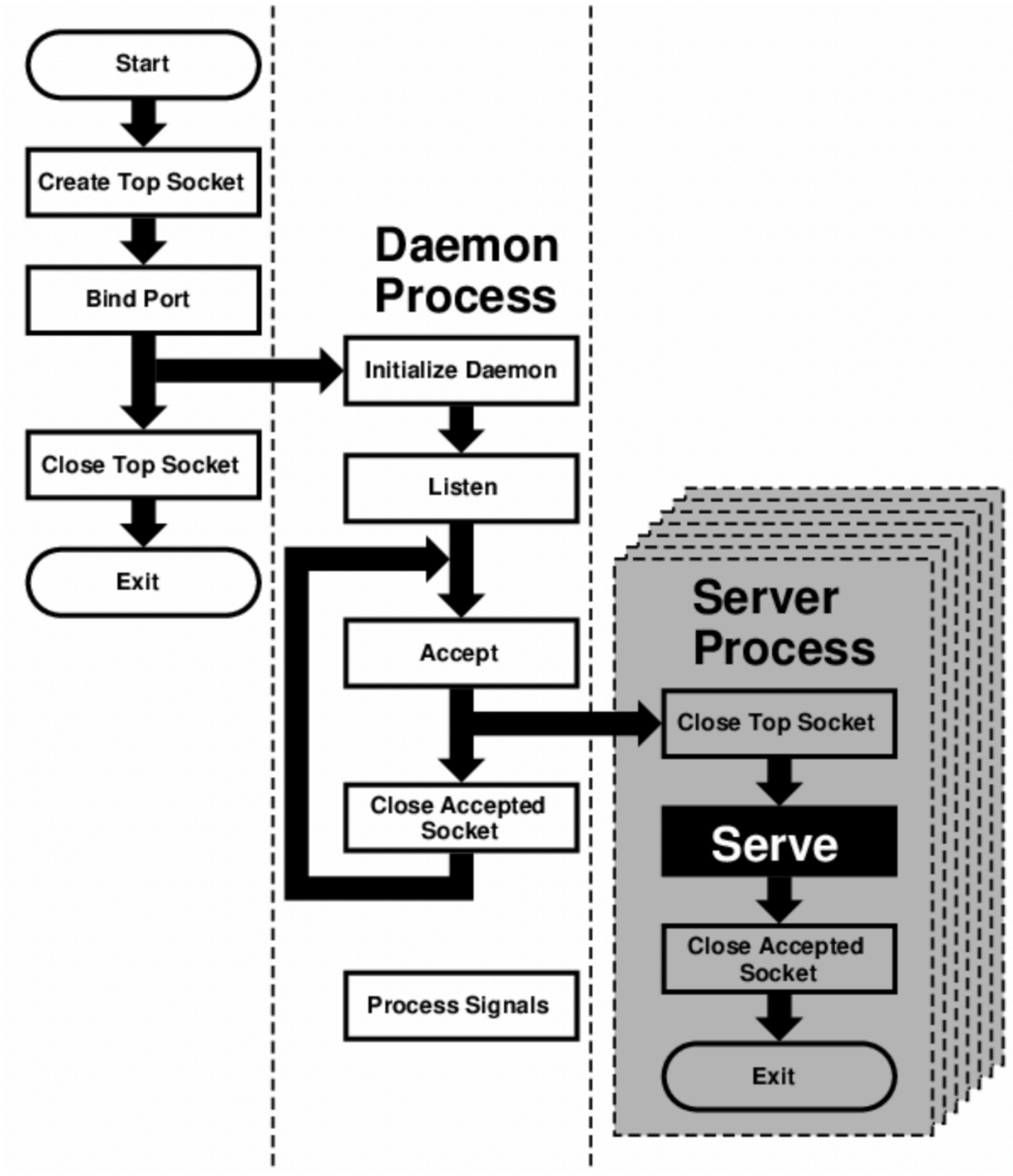
https://docs.freebsd.org/en/books/developers-



Concurrent Server

from

https://docs.freebsd.org/en/books/developers-handbook/sockets



UDP client e UDP server

UDP workflow

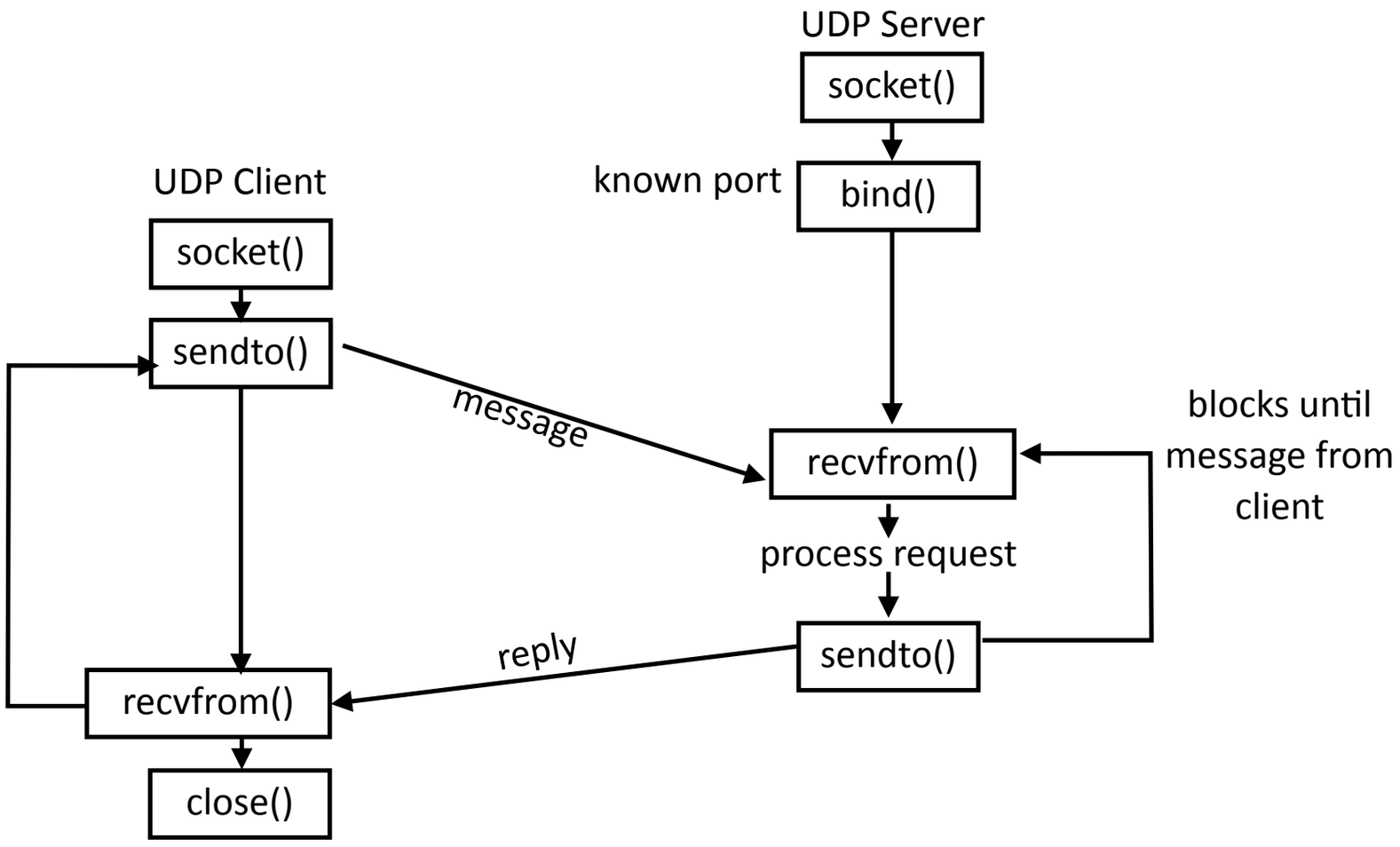
Catiuscia Melle and Franco Tommasi - SNP 2021-2022

UDP communications

- Unlike the TCP protocol, **UDP** does not need a connection between two remote node to be established to allow them to exchange information.
  - *connectionless protocol*
  - clients are not required to connect to the server
  - server is not required to accept connections from clients
- All it takes for the server is to be listening a *UDP well-known port*
  - the server creates a socket and associates it to the service port (calling ***bind()***)
  - clients have only to send messages to the known address and wait for an answer
- Since UDP sockets are not connected, with the same *socket descriptor* it is allowed:
  - to send a message to different destinations
  - to receive message from different peers
  - *many-to-many communications*

UDP communications

- The typical UDP workflow (*timeline*) is shown in the picture:



- As for a TCP server, a UDP server must call *bind()* specifying the *well-known address* by which it may be reached

UDP communications

- *sendto()*: parallels *send()*, but here the destination (*socket address*) must be provided
  - similarly to the *connect()* API

```
#include <sys/socket.h>

ssize_t sendto(int sock, const void *message,size_t len,int flags,
               const struct sockaddr *dest_addr, socklen_t dest_len);
```

- *sendto()* returns the number of written bytes, or -1 in case of error
  - Contrary to TCP, it is possible to send *0 byte* messages:
    - They will be datagrams with a *20 byte IP header* + an *8 byte UDP header an no UDP payload*
      - POSIX advises to avoid it (some implementations do not support it)
- Contrary to TCP, a message may be sent to more different destinations using the same socket descriptor and each time specifying a *different* destination
  - a TCP socket is *connected* to a single peer

UDP sendto

- Unlike TCP, a UDP socket do not have a Send Buffer
  - Since UDP is unreliable, the kernel doesn't need to keep a copy of the datagrams sent by the application, so there is no send queue
    - The UDP layer has no retransmission procedure
  - the message coming from the application is copied into a kernel buffer and discarded as soon it it sent
- the kernel limits the *maximum size* of the UDP datagram written to the socket
  - If the application writes a datagram greater than the allowed size, the kernel returns the **EMSGSIZE** error
- **sendto()** returns positively only when the datagram has been *queued* in the datalink layer output queue
  - If the output queue has not enough space to queue datagram, the **ENOBUFS** error is returned to the process
    - Not all implementations return such error: some simply discard the datagram without informing the application the datagram has been not even queued (therefore not transmitted)

UDP communications

- **recvfrom()**: similar to *recv()*, with the difference the message sender is not known a priori, but can be obtained passing *address* and *address_len* to the call

```
#include <sys/socket.h>
ssize_t recvfrom(int sock, void *restrict message,size_t len,
                 int flags, struct sockaddr *restrict address,
                 socklen_t *restrict address_len);
```

  - analogously to *accept()*: *address* and *address_len* are *value-result arguments*

- In case of success, *recvfrom()* returns the read ***datagram*** *length*
  - *-1* in case of error
  - the received UDP datagram payload is stored in *message*
  - up to a maximum size of *len*: the remaining part is discarded
    - Unlike TCP which support a byte-stream
    - Unlike TCP, the reception of a *0 byte* message has no special meaning
  - Infos on the message sender are in *address*
    - comparing *address* with the address used in a previous *sendto()* the process may verify the sender is the same peer and possibly discard undesired traffic

UDP unreliability

- Like previous system calls *sendto()* e *recvfrom()*, are ***blocking***
  - Notably, a process may remain blocked *indefinitely* in a *recvfrom() call,* should the expected datagram be lost/discarded without being retransmitted
- Similarly, a client sending a message to a *not running* UDP server, will remain blocked forever in the *recvfrom(),* waiting for an answer that will **never** arrive
  - Using a sniffer, an *ICMP port unreachable message* sent to the client process can be observed
  - Such ICMP error message is an **asynchronous error message**, since i is caused by *sendto()*, but *sendto()* returned successfully
    - the datagram queued in the output queue of the datalink layer
    - the ICMP error message appeared only after forwarding operations were concluded
    - In general, an asynchronous error *is never returned* to a UDP socket, unless the socket is *connected*

Connected UDP Sockets

- In general, the sending functions - *send()* and *sendto()* - generate ***asynchronous*** errors
  - An error related to a datagram transmission is generated only after data have been delivered to the link (and to the network)
  - *ICMP destination unreachable* messages are received only after an attempt to send a datagram to an *unreachable destination is done.*
- A default UDP socket will never receive feedback about asynchronous errors, unless the UDP socket becomes ***connected***
  - In such case the kernel will return the error to the socket so that the process may detect it.
- A default UDP socket is *not connected*
- It is possible to use ***connect()*** with a UDP socket:
  - Obviously that doesn't start neither a 3WHS as in TCP
  - Nor the activation of datagram retransmission mechanisms

Connected UDP Sockets

- The effect of *connect()* for a UDP socket is to *restrict the communication of the socket to a single peer*
- The kernel records in the socket the information about the remote peer
  - The remote address is given as an input to *connect()*
  - The *socket pair* is now entirely defined *(remote address, local address)*
- At this point it is possible to use *send()* and *recv()* instead of *sendto()* and *recvfrom()*
  - The kernel has already stored in the socket the relevant information so it doesn't need it
  - Messages coming from other addresses are at this point discarded and the kernel send an *ICMP port unreachable* message to the peers
- **only messages coming from the connected peer are delivered to the application**

Connected UDP Sockets

- An UDP connection make sense *only when* restricting the communication with a *single peer* is preferred
- A connected UDP socket increases the application's security
  - Only the traffic coming from the connected peer is received by the application
- The execution of a *connect()* for an UDP socket results in:
  - the kernel consults the node routing table and selects the outgoing interface to be used for that particular destination
  - such information is stored in the socket's local address field
- On return form a *connect()*, it is possibile to know *local IP and port* assigned from the kernel to the socket - to communicate with the connected peer - by *getsockname()*
  - Same as with a TCP socket

Connected UDP Sockets

- Some platforms implement *sendto()* as a wrapper to *send()*:
  - the socket is connected to the destination specified by *sendto()* and *send()* is used for the actual transmission
- A connected UDP socket offers some performance gains to the application:
  - there is no need to process the destination address and to scan the node routing table: once the decision is taken at the connect() call it is stored and used for the following datagrams
    - On Linux, the use of connected UDP socket does not offer performance gains

Disconnection of UDP Sockets

- *connect()* may be used more than one for an UDP socket
  - To specify a new address:
    - Contrary to *TCP* for which *connect()* may be invoked only once, we may connect the socket to a new peer
  - or to disconnect the socket
- The socket disconnection is *platform-dependent*:
  - Stevens suggests the most portable solution is *to zero* the address structure, set *AF_UNSPEC* as address family and give it as an input to *connect()*
  - The socket *unconnect* procedure differs in systems also with respect to local binding:
    - in particular, it is not guaranteed the socket will keep the local IP address it had when *connected* (it usually keeps the port binding instead)

TCP reliability

TCP 4 way teardown

Reliable data transfer among two remote nodes while closing a TCP connection

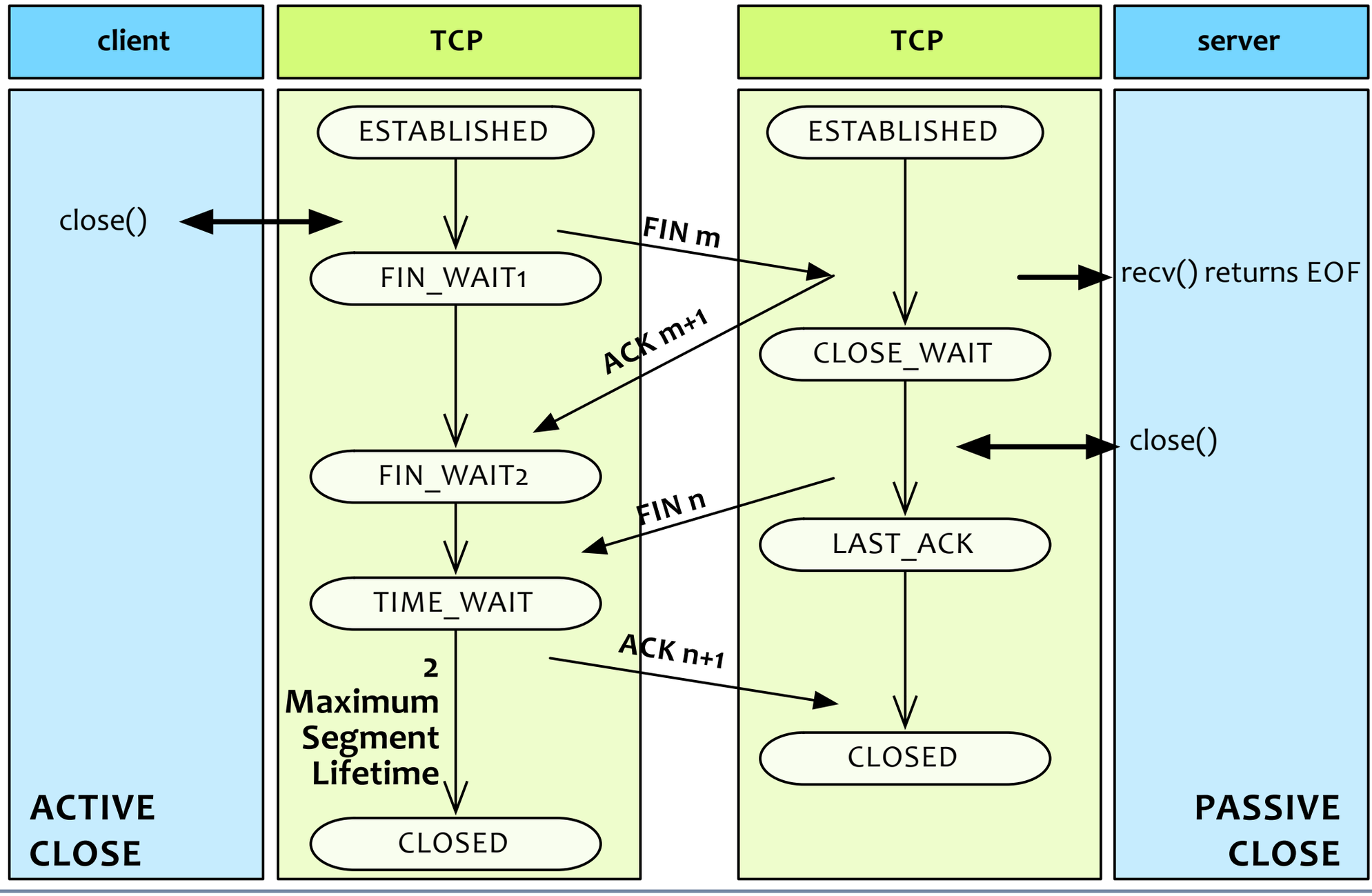
Catiuscia Melle and Franco Tommasi - SNP 2021-2022

TCP 4-way teardown

- The TCP protocol requires the termination of a connection to be performed by a procedure guaranteeing a correct behaviour at both endpoints
  - ***3*** segments are needed to establish a TCP connection, ***4*** segments are needed to terminate it
- The TCP protocol distinguishes the two endpoints' roles:
  - The one performing a *close()* first does an ***active close***
    - it has no more data to send thus terminates the connection
    - the application calls *close()* and the TCP layer sends a **FIN** to the other endpoint
  - The one receiving the **FIN**, does the ***passive close***
    - TCP acks the *FIN* as soon as it receives it
    - The *FIN* reaches the application as *End-of-File*, ***after all data already queued in the*** **socket receive buffer**
- When the app receives the EoF (*recv() returns 0*) becomes aware it will not receive any more data and closes its side of the connection
  - Now the *passive side* sends a **FIN**
  - The *active side* must wait, to ack the *passive side FIN*
- Since TCP must guarantee reliability also during the close, The *active side* must *persist until it makes sure the passive side has concluded its operation*
  - Since the two sides play different roles, TCP assigns different state transitions to each of them

TCP 4-way teardown

- The *active TCP endpoint* enters the **TIME_WAIT** state to allow the *passive TCP endpoint* to receive the final *ACK* (for its *FIN*)



TIME_WAIT and Maximum Segment Lifetime

- The duration of the **TIME_WAIT** state is **twice** the **Maximum Segment Lifetime** (*MSL*)
- Each TCP implementation has its own *MSL* value
  - The recommended value is ***2 minutes*** (RFC 1122),
  - BSD-derived implementations and Linux have an MSL value of *30 seconds*
    - This means the duration of the TIME_WAIT state goes from 1 to 4 minutes
- **MSL** is based on a rough estimate of the maximum time an IP datagram may exist in the network
- The TIME_WAIT state exists for two reasons:
  - to implement in reliable way the closing of a TCP connection
  - to allow the old segment to “expire”

TIME_WAIT state

- An IP datagram may get lost in the network because of routing anomalies
  - router crashes, link faults,…
  - routers look for *alternative paths in such* cases
  - this causes a delay of the packet delivery or the datagrams loss for expired TTL
- When a *lost IP datagram* contains a *TCP segment*, the TCP sender retransmits it after timeout is expired
  - this in turn may imply the arrival of *multiples copie* of the same message
    - the receiving TCP will discard them
- The *TIME_WAIT* state guarantees **all** TCP segments (*passive endpoint FIN* included*)* are ACKed before the connection is closed,
  - If TIME_WAIT were *1 MSL*, the reception of the final ACK by the *passive endpoint* wouldn't be guaranteed
    - 1 MSL to wait for the last transmitted ACK reaches the *passive endpoint*
    - 1 MSL to receive the *passive endpoint* FIN retransmission (if the previous I'ACK got lost)

TIME_WAIT state

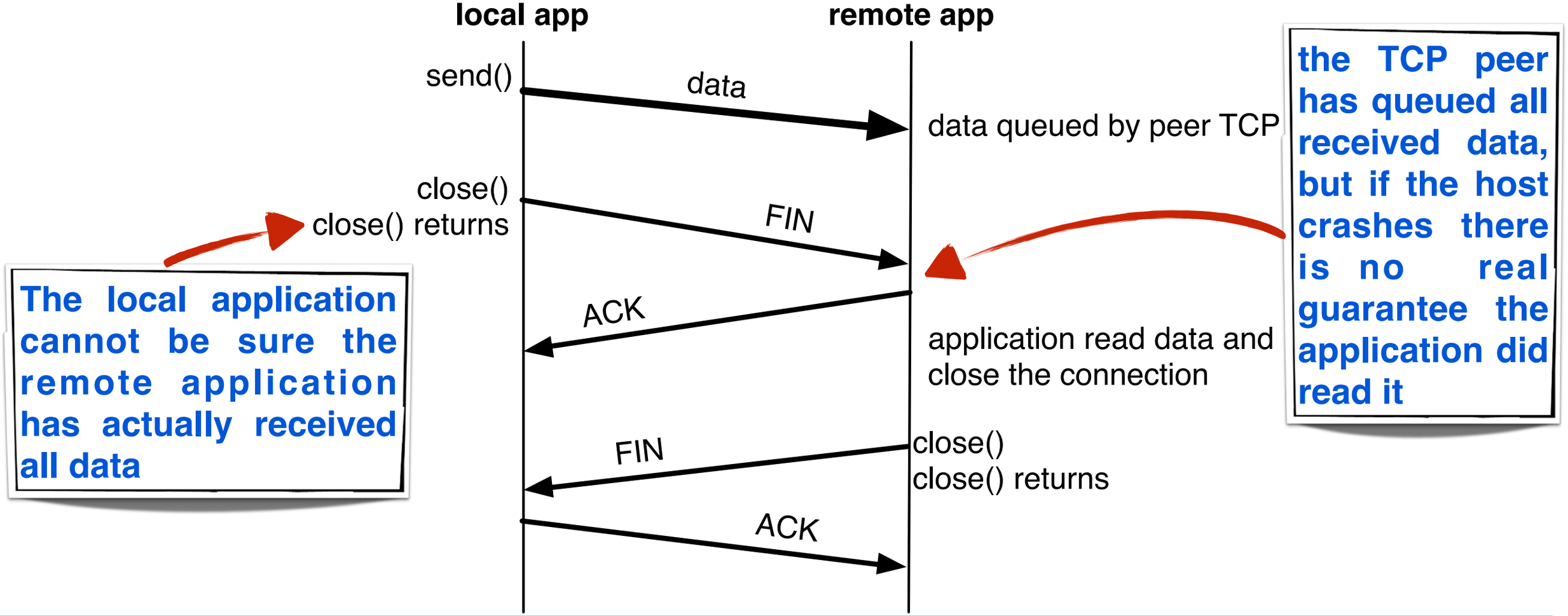
- Beside, the TIME_WAIT state prevents a ***new incarnation*** to be started if all old segments (from the previous connection) are not yet expired in the network
  - *New incarnation:* a new connection started with the same socket pair of the previous TCP connection
  - In that case, old segments could (erroneously) play a role in the new connection
- The attempt to start a new connection when one of the endpoints is in the *TIME_WAIT* state, causes the error **EADDRINUSE** to be returned for *bind()*
    - Actually, simply choosing sequence numbers greater than the last used for the new connection SYN would prevent any such problem

Closing a connection

- When one of the two processes calls *close(),* both direction of the connection are closed
  - No further read of the received data is allowed
  - No further write to the socket is allowed
- This actually means, the reliableness of the data exchange between the two endpoints *is not guaranteed*
  - If data are present in the socket RECV Queue, the process will not be allowed to read them after the socket closure, and TCP may discard them and the possible following data
- An application protocol should be able to monitor such situations and to guarantee a reliable communication also in the closing phase.

TCP connection closing: close()

- The local application closes the TCP connection calling *close()*
  - The local TCP is responsible for the data *reliable delivery* to the remote peer
  - The local application cannot be sure the remote application has actually received all data



TCP connection Half-Closing

- *shutdown()* allows to close a single direction:

```
#include <sys/socket.h>

int shutdown(int socket, int how);
```

  - The *how parameter* defines how the connection is closed, disabling following send() or receive() calls on the socket

- Unlike *close()*, *shutdown()* acts on the socket *independently* from the *reference count* (e.g. a child process may have the same socket open)

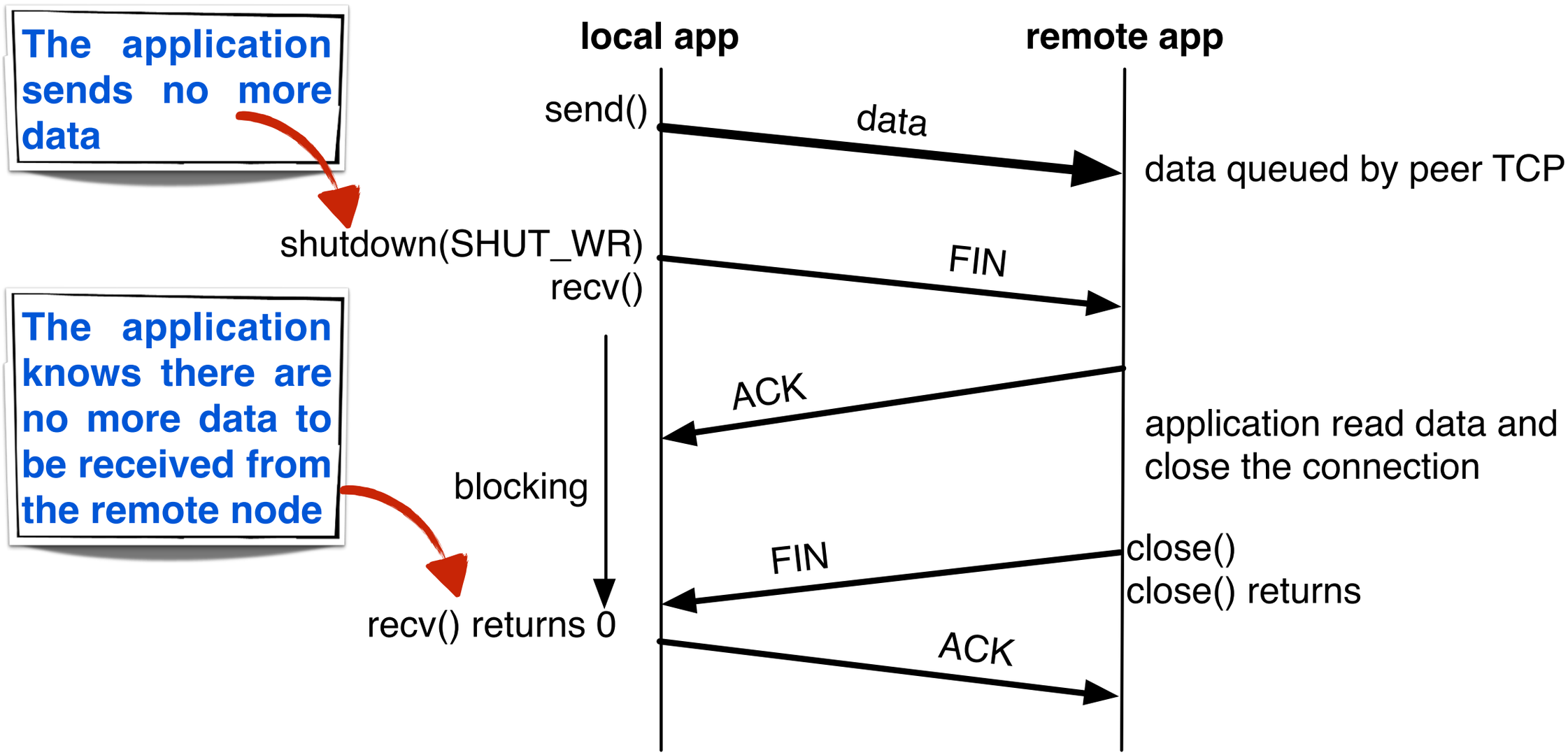
- Beware: to deallocate the socket (that is, the resources allocates by the kernel) the *close()* call **is** required anyway

TCP connection Half-Closing

- how = **SHUT_RD**
  - No more receive operations allowed
    - no more data received from the socket. What is in the socket receive buffer is *discarded*
- how = **SHUT_WR**
  - No more send operations allowed
    - TCP calls this ***half-close***: data in the socket send buffer will be delivered, then the TCP closing procedure will start
- how = **SHUT_RDWR**
  - No more send and receive operations allowed
    - Same as calling shutdown twice, first with SHUT_RD then with SHUT_WR

TCP connection Half-Closing

- Thanks to the *shutdown()* call, the application may guarantee the complete data exchange: when receiving the EOF from remote node, the local will know all its data have been **read** by the remote application



I/O flags

Per-call flags

Catiuscia Melle and Franco Tommasi - SNP 2021-2022

I/O operation flags

- *flags is an i*nput parameter of *recv()/recvfrom()* and *send()/sendto()*
- *flags* is a bit mask by which the caller can modify the behavior of a single socket I/O call
- Its values are defined by the SuSv4 standard
  - More values can be ORed
- Each O.S. allows a set of non standard values
  - *not portable*
- For example, POSIX defines the **MSG_PEEK** flag for reading operations:
  - if **MSG_PEEK** is set in *recv()/recvfrom()*, the kernel allows the application to read the data in the socket RECV Queue but doesn't remove them.
  - read data are left available for the following reading

I/O flags defined in SuSv4

- recv() e recvfrom() flags

MSG_PEEK: Peeks at an incoming message. The data is treated as unread and the next recv() or similar function shall still return this data.

MSG_OOB: Requests out-of-band data. (The significance and semantics are protocol-specific).

MSG_WAITALL: On SOCK_STREAM sockets this requests that the function block until the full amount of data can be returned. The function may return the smaller amount of data if the socket is a message-based socket, if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket.

- send() e sendto() flags

MSG_EOR: Terminates a record (if supported by the protocol).

MSG_OOB: Sends out-of-band data on sockets that support out-of-band data.
The significance and semantics of out-of-band data are protocol-specific.

MSG_NOSIGNAL: Requests not to send the SIGPIPE signal if an attempt to send is made on a stream-oriented socket that is no longer connected. The [EPIPE] error shall still be returned.

  - In Mac OS X, MSG_NOSIGNAL is not defined, but in <sys/socket.h> we find

```
#define SO_NOSIGPIPE  0x1022    /* APPLE: No SIGPIPE on EPIPE */
```

I/O operation flag

- The standard defines the **MSG_NOSIGNAL** flag for writing operations:
  - If a writing is attempted on a no more connected socket, the set flag asks not to generate the **SIGPIPE** signal (the default) but to set errno to **EPIPE** instead
- As an example of a non standard flag, we may cite the **MSG_DONTWAIT** flag:
  - implemented in Linux and Mac OS X
  - If an I/O operation with a socket uses the **MSG_DONTWAIT** flag, the operation will be **not blocking**
  - The call will the return either with *success* or with ***-1*** and *errno* equal to *EWOULDBLOCK* or *EAGAIN* to signal the interruption of the operation
  - Unlike other non blocking modes for a socket, defined for all socket operations (socket timeout or non-blocking mode), the use of the flag is on a per-call base.

Hostname Resolution

Domain Name System

Catiuscia Melle and Franco Tommasi - SNP 2021-2022

Service and host names

- Starting from human-readable numerical addresses, s*truct sockaddr_in/ sockaddr_in6* have been initialized, in Network Byte Order, with
  - **inet_pton()** for the host address part
  - and with **htons()** for the service port part (TCP/UDP port)
- It is also possible to provide a *hostname* and a *servicename* to indicate the remote host and the service port
  - names instead of numbers:
    - Example: *hplinux2.unile.it:http* instead of *193.204.77.157:80*
- A way to convert i *names* in *numerical values (*essential to initialize protocol addresses used in Socket API) is needed: ***Domain Name System***
- The task of DNS is to map hostname to IP addresses and viceversa
  - The most widely known and used implementation of a *Domain name server* is the *BIND* program (**do not confuse it with bind() !!**)
    - **B**erkeley **I**nternet **N**ame **D**omain [Albitz and Liu]

Service and host names

- When Internet started, since hosts were few, mapping between hostnames and IP addresses was kept in the local file **/etc/hosts**

```
#IP-address hostname  [canonical] [aliases]
127.0.0.1    localhost
```

- Network applications looked up IP addresses corresponding to a given hostname in the local file
  - Obviously this solution have scalability problems:
    - what happens when nodes number increases?
    - what happens when mapping has to be modified (globally)?
- DNS solved these problems by organizing names in a *hierarchical namespace*, partitioned in *zones*
  - each node has a *name* (less then 63 characters)
  - the *root node* of the hierarchy is the ***anonymous root "."***

Resolver

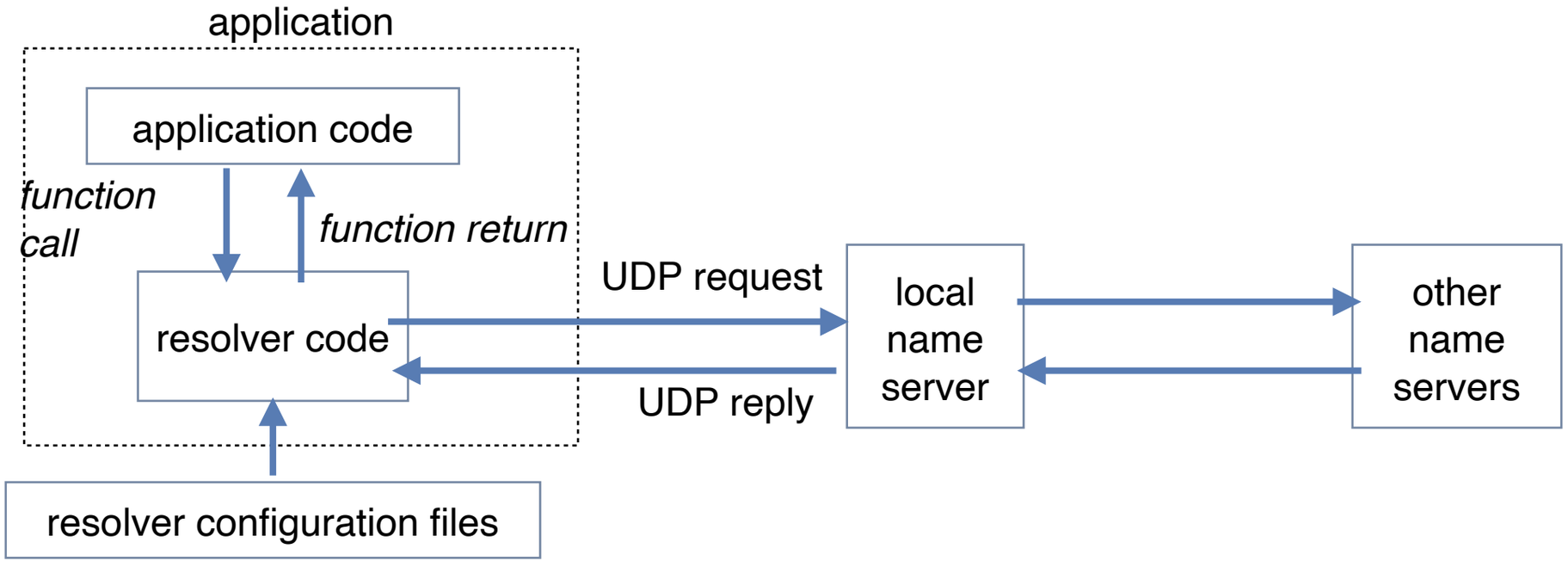
- A network application contacts the DNS server to obtain an hostname resolution, by a library function called ***resolver***
- The resolver, thanks to parameters contained in a local file, is able to locate the DNS server to contact.
  - *IP addresses* of DNS servers a machine has to use can be found in ***/etc/resolv.conf***

```
nameserver 8.8.8.8
```

- The resolver sends a *query* to a local DNS server (which, in turn, may decide to forward it to other DNS servers)
  - when the resolution process terminates, the DNS server will send back to the resolved the answer, that is the conversion of the original hostname into the corresponding IP address (or the pool of IP addresses associated to that hostname)
- The resolver's API is the ***getaddrinfo()*** call, which substituted the historical *gethostbyname()* (deprecated and removed from SuSv4)

Resolver

- If the queried DNS server doesn't know the answer, the query is forwarded to other DNS servers:
  - *recursively*, if the queried DNS server is responsible for the answer fulfillment
  - *iteratively*, if the queried DNS server gives back the IP address of another DNS Server



- a hostname may be a *simple name* (es: **liis1**) or a *Fully Qualified Domain Name* - ***FQDN*** - also known as *absolute name* (example: *liis.unile.it*)
  - complete *FQDN*s end with a "." which is the *anonymous root*
  - the *dig* command allows to interrogate a DNS server; the *whois* command allows to get information about names and addresses

Domain Name System

- DNS entries are called *Resource Records* (*RR*)
- RRs an application usually queries about are:
  - **A**: maps an *hostname* to an *IPv4* address
  - **AAAA** (*quad A*): maps an *hostname* to an *IPv6* address
    - *quad A* because an IPv6 address 4 times longer than an IPv4 address
  - **CNAME** (*Canonical Name*): an *alias* of the main *hostname*
  - **PTR (Pointer Records)**: map an *IP address* into the corresponding *hostname*
- Example: RRs A ed AAAA for an hypotetical *liis1 host* would be:

```
liis1     IN A     12.106.32.254
          IN AAAA  3ffe:b80:1f8d:1:a00:20ff:fea7:686b
```

  - PTR records are the IP addresses, in reverse order, that is, for the above host:

    `254.32.106.12.in-addr.arpa` for *IPv4*

    `b.6.8.6.7.a.e.f.f.f.0.2.0.0.a.0.1.0.0.0.d.8.f.1.0.8.b.0.e.f.f.3.ip6.arpa` for *IPv6*

Names resolution

- The conversion functions removed from the SuSv4 standard, were:
  - *gethostbyname()*: performed a DNS query to resolve a hostname into an IPv4 address (*no IPv6*)
  - *gethostbyaddr()*: the inverse function, starting from a numerical IPv4 address (contained in a *in_addr struct*) returned the corresponding hostname
- Another couple of functions resolved service names in both directions:
  - *getservbyname()* and *getservbyport()*, worked by reading the file **/etc/services**
- These functions are now declared obsolete:
  - mainly because they do not support IPv6 addresses
  - also because they are not *re-entrant*

Names resolution

- The new function for names resolution is **getaddrinfo()**
- **getaddrinfo()**:
  - it supports names resolution both for IPv4 and IPv6
  - it resolves also service names (in port numbers)
  - it is reentrant
  - it allows to write *protocol-independent code*

getaddrinfo

- ***getaddrinfo()*** does a DNS query (*when needed*) and returns an integer:
  - ***0*** in case of success,
    - in such case, *res* is the pointer to a linked list of ***struct addrinfo*** containing al requested information
    - a *struct addrinfo* contains a *struct sockaddr,*
      - which can be directly used in following *socket()*, *bind()*, *connect()*, *sendto()* calls
  - **!=0** in case of error

```
#include <sys/socket.h>
#include <netdb.h>

        int getaddrinfo(const char *restrict nodename,
                        const char *restrict servname,
                const struct addrinfo *restrict hints,
                        struct addrinfo **restrict res);
```

getaddrinfo()

- **getaddrinfo()** input parameters represent:
  - **nodename**
    - may be an *hostname* (es: mach3.unile.it) or an address string (dotted-decimal for IPv4, hexadecimal string for IPv6)
  - **servname**
    - may be a service name (e.g. http,..) or a *string containing the port decimal value* (e.g. 80)
  - **hints**
    - may be *NULL* or the address of an *addrinfo struct* initialized with the parameters the application wishes to use for the resolution
      - parameters to be provided to the resolver to build the DNS query
  - **res**
    - value-result argument: a pointer to the linked list of *struct addrinfo*, allocated and returned by the function after the resolution is completed

struct addrinfo: definition

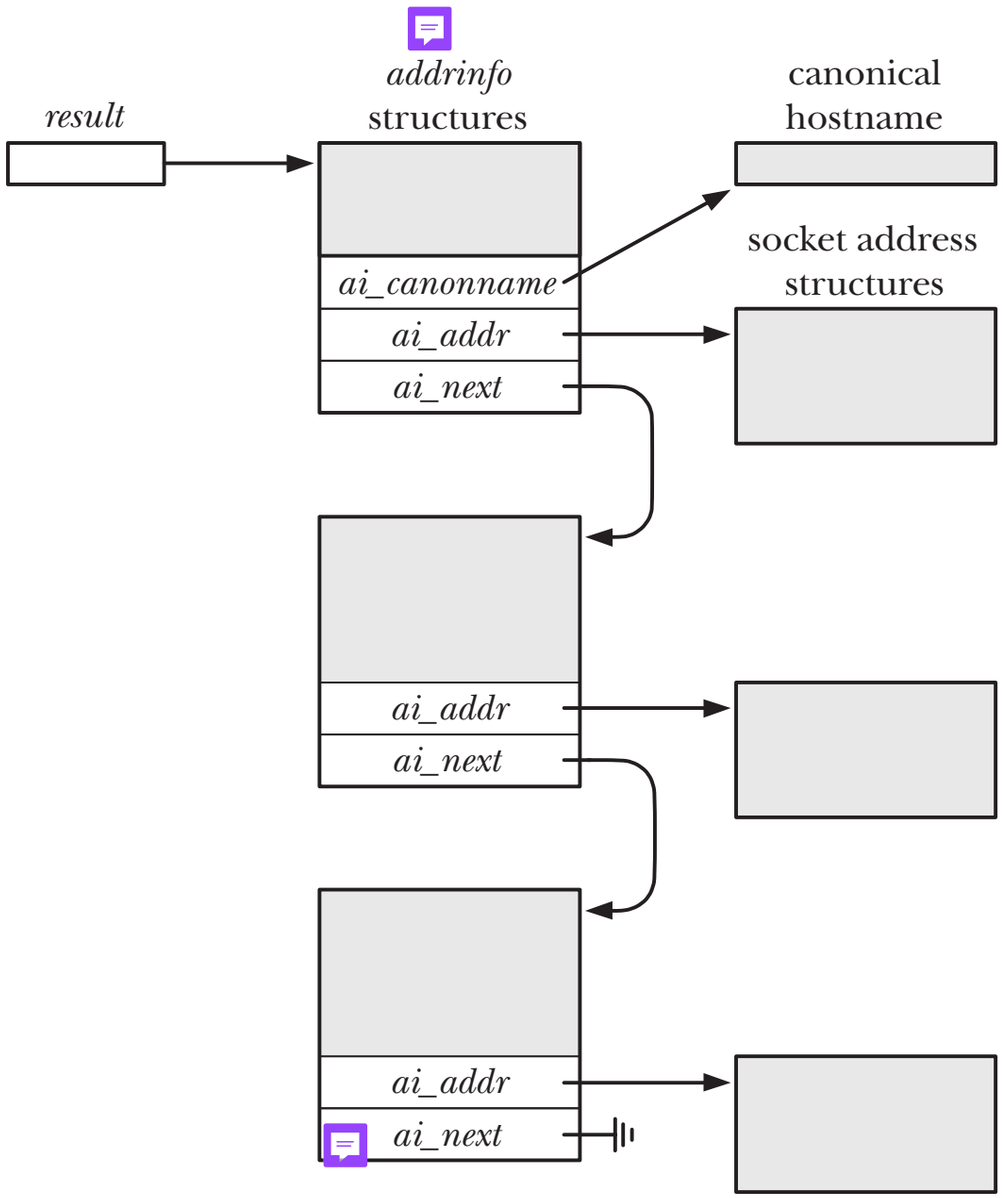
- *struct addrinfo* is defined as following:

```
struct addrinfo{
int              ai_flags      //Input flags
int              ai_family     //Address family of socket
int              ai_socktype   //Socket type
int              ai_protocol   //Protocol of socket
socklen_t        ai_addrlen    //Length of socket address
struct sockaddr  *ai_addr      //Socket address of socket
char             *ai_canonname //Canonical name of the hostname
struct addrinfo  *ai_next      //Pointer to next in list
};
```

- When *struct addrinfo* plays the role of an input parameter for *getaddrinfo()*, it allows to specify the resolution parameters:
  - ai_flags
  - ai_family
  - ai_socktype
  - ai_protocol

getaddrinfo() result

- In case of success, *getaddrinfo()* returns a linked list of *struct addrinfo*
- Each item in the list pointed by *res*, will satisfy the requirements hinted in *hints*
  - If *canonical name*, have been requested it will be returned only in the first element of the list.
  - Each *struct addrinfo* contains a *struct sockaddr*, pointed by *ai_addr*, set to the IP address and the port corresponding to the hostname and servicename given as input the al resolver
  - *ai_addrlen* contains the size of the corresponding allocated *struct sockaddr*
  - *ai_next* points to the next item of the list (or *NULL* in the last item)



struct addrinfo and the hints

- **hints** fields the caller may define are:
  - **ai_flags**: 0 or more ORed values (**I**)
  - **ai_family**: allows to restrict the Address Family of the address returned in the sockaddr field to a specific value (AF_INET, AF_INET6, AF_UNSPEC)
    - AF_INET for *struct sockaddr_in*, AF_INET6 for *struct sockaddr_in6*
    - According to the standard, by *AF_UNSPEC*, the resolver will return both *AAAA records* and *A records* (in *struct sockaddr_in6* and e *struct sockaddr_in accordingly)*
  - **ai_socktype**: the type of socket the address structures will be used for (SOCK_DGRAM, SOCK_STREAM):
    - a 0 value accept both services
  - **ai_protocol**: same as the above (IPPROTO_TCP or IPPROTO_UDP, 0 indicates all protocols are accepted)

struct addrinfo and the hints

- *hints.ai_flags* influences the behaviour of *getaddrinfo()*
  - it is a binary mask which controls the resolution process
- May be set using a combination of the following constants:
  - **AI_PASSIVE**: the caller wishes the socket for a Passive Open - that is a following call to *bind()*
    - If not set, the returned addresses will be used in following *connect/sendto (active open)*
  - If specified with *nodename=NULL*, then the returned addresses will be set to **INADDR_ANY** for IPv4 and **IN6ADDR_ANY_INIT** for IPv6
  - if (*nodename=NULL)* & (AI_PASSIVE not set), it will return the loopback address

ai_flags

- **AI_CANONNAME**: asks for the *hostname canonical name*
  - saved in the field *ai_canonname* of the first *struct addrinfo* pointed by *res*
    - if the host has no canonical name, a copy of nodename will be returned
- **AI_NUMERICHOST**: if set, it means *nodename is an address string*
  - no need to start the Resolver and a DNS query
  - conversion is then managed by *inet_pton()*, specifying the required address family
- **AI_NUMERICSERV**: if set prevents the service name resolution
  - service is already a string containing the decimal value of a port address (16 bit)
    - if flags *AI_NUMERICHOST* and *AI_NUMERICSERV* are set but the input strings are not **valid addresses or ports**, then the error **EAI_NONAME** is returned

ai_flags

- **AI_V4MAPPED**: if used with *ai_family=AF_INET6*, then *getaddrinfo()* will return the ***IPv4-mapped-IPv6*** address corresponding to the A records
  - if *AAAA records for the specified hostname are not available*
    - If AI_V4MAPPED is not used with ai_family=AF_INET6, it is ignored
- **AI_ALL**: if used with **AI_V4MAPPED**, makes *getaddrinfo()* return, all *AAAA records*, **and** *the IPv6 IPv4-mapped-IPv6 addresses corresponding to A records associated to the hostname*
- **AI_ADDRCONFIG**: return IPv4 addresses only if there is at least one IPv4 address configured for the local system (other than the IPv4 loopback address), and return IPv6 addresses only if there is at least one IPv6 address configured for the local system (other than the IPv6 loopback address).

freeaddrinfo()

- *getaddrinfo()* allocates the memory needed for the linked list returned to the application
  - The application will have to deallocate all no more needed structures
- *freeaddrinfo()* takes as input the pointer to the linked list of *struct addrinfo* and deallocates the busy memory in a single step

```
#include <netdb.h>
#include <sys/socket.h>

void freeaddrinfo(struct addrinfo * res);
```

- After deallocation the structures will be no more accessible:
  - The application will have to make a local copy of the addresses of interest and then call *freeaddrinfo()* for the linked list

getnameinfo(): reverse lookup

- *getnameinfo()* is the inverse function of *getaddrinfo()*: it allows to get hostname and service name corresponding to numerical values present in an address structure
  - If names cannot be resolved, the function returns the numerical representation

```
#include <netdb.h>
#include <sys/socket.h>

int getnameinfo(const struct sockaddr *addr, socklen_t addrlen, char
*host, size_t hostlen, char *service, size_t servicelen,int flags);
```

- *addr* points to the *struct sockaddr* whose reverse lookup is requested, of size *addrlen*
- hostname and service will be returned in strings returned by *host* and *service* (lengths *hostlen* and *servicelen*)
  - The standard does not define the length of the two strings, but some implementations define in *<netdb.h>* ***NI_MAXHOST*** *(1025)* and ***NI_MAXSERV*** *(32) (by setting appropriate feature test macros)*

getnameinfo()

- *getnameinfo()* operates in a **protocol-indipendent** way, on *struct sockaddr*, transparently with the IP version (no explicit dependency from the Address Family)
  - *host* and *service* strings must be allocated by the caller
    - if *hostlen=0*, the call will not return host information
    - if *servicelen=0*, the call will not return service information
      - **they cannot be both null at the same time**
- the *flags* parameter allows to control the behaviour of *getnameinfo() during the* reverse lookup, by *ORing the following constants*:
  - **NI_DGRAM**: for an UDP service
    - usually the same port number is assigned to the TCP and UDP version of the same service: if that is not the case such flag forces the UDP service to be returned
  - *NI_NAMEREQD*: if the hostname cannot be resolved, the error EAI_NONAME is returned instead of the numeric representation
  - *NI_NOFQDN*: asks for the hostname part of the name instead of the full DNS name
  - **NI_NUMERICHOST**, **NI_NUMERICSERV**: force numeric representations to be returned and disable both the DNS query and the */etc/services* file reading

gai_strerror()

- In case of error, *getaddrinfo()* e *getnameinfo()* return a value !=***0***
- The function ***gai_strerror()*** interprets the *getaddrinfo()* and *getnameinfo()* functions errors, and returns a string describing the error

```
#include <netdb.h>
#include <sys/socket.h>

const char *gai_strerror(int ecode);
```

Error constant	Description
EAI_ADDRFAMILY	No addresses for *host* exist in *hints.ai_family* (not in SUSv3, but defined on most implementations; *getaddrinfo()* only)
EAI_AGAIN	Temporary failure in name resolution (try again later)
EAI_BADFLAGS	An invalid flag was specified in *hints.ai_flags*
EAI_FAIL	Unrecoverable failure while accessing name server
EAI_FAMILY	Address family specified in *hints.ai_family* is not supported
EAI_MEMORY	Memory allocation failure
EAI_NODATA	No address associated with *host* (not in SUSv3, but defined on most implementations; *getaddrinfo()* only)
EAI_NONAME	Unknown *host* or *service*, or both *host* and *service* were NULL, or AI_NUMERICSERV specified and *service* didn't point to numeric string
EAI_OVERFLOW	Argument buffer overflow
EAI_SERVICE	Specified *service* not supported for *hints.ai_socktype* (*getaddrinfo()* only)
EAI_SOCKTYPE	Specified *hints.ai_socktype* is not supported (*getaddrinfo()* only)
EAI_SYSTEM	System error returned in *errno*

IPv4-IPv6 Interoperability

source code portability

Catiuscia Melle and Franco Tommasi - SNP 2021-2022

getaddrinfo(): results and actions

- If Address Family is *AF_UNSPEC*, the resolver returns both *AAAA* and A Resource Records
  - IPv6 addresses come first

Hostname specified by caller	AF	Hostname string contains	Result	Action
not null hostname string; active or passive	**AF_UNSPEC**	**hostname**	**all AAAA** records returned as *sockaddr_in6* ***and*** **all A** records returned as *sockaddr_in*	AAAA RR **and** A RR DNS search
		hex string	one **sockaddr_in6**	inet_pton(AF_INET6)
		dotted decimal	one **sockaddr_in**	inet_pton(AF_INET)

getaddrinfo(): results and actions

- If Address Family is *AF_INET6*, the resolver returns only *quad-A* records
  - results may vary according to conditions and flags as shown below

Hostname specified by caller	AF	Hostname string contains	Result	Action
not null hostname string; active or passive	**AF_INET6**		**all AAAA** records returned as *sockaddr_in6*	AAAA RR DNS search
		hostname	if *ai_flags* contains **AI_V4MAPPED**, **all AAAA** records as *sockaddr_in6* **else all A** records as IPv4-mapped-IPv6 in *sockaddr_in6*	AAAA RR DNS search if no results: A RR DNS search
			if *ai_flags* contains **AI_V4MAPPED and AI_ALL**, **all AAAA** records as *sockaddr_in6* **and all A** records as IPv4-mapped-IPv6 in *sockaddr_in6*	AAAA RR and A RR DNS search
		hex string	one **sockaddr_in6**	inet_pton(AF_INET6)
		dotted decimal	looked up as hostname	

getaddrinfo(): results and actions

- If Address Family is *AF_INET*, the resolver returns only *A* records

Hostname specified by caller	AF	Hostname string contains	Result	Action
not null hostname string; active or passive	**AF_INET**	hostname	all A records returned as *sockaddr_in*	A RR DNS search
		hex string	looked up as hostname	
		dotted decimal	one **sockaddr_in**	inet_pton(AF_INET)

getaddrinfo(): results and actions

- *Passive Open*
  - hostname must be *NULL* (no lookup performed)
  - AI_PASSIVE flag must be set

Hostname specified by caller	AF	Hostname string contains	Result	Action
NULL hostname string; passive	**AF_UNSPEC**	implied :: implied 0.0.0.0	one **sockaddr_in6** **and** one **sockaddr_in**	inet_pton(AF_INET6) inet_pton(AF_INET)
	AF_INET6	implied ::	one **sockaddr_in6**	inet_pton(AF_INET6)
	AF_INET	implied 0.0.0.0	one **sockaddr_in**	inet_pton(AF_INET)

getaddrinfo(): results and actions

- *Active Open*
  - hostname must be *NULL* (no lookup performed, loopback address returned)
  - AI_PASSIVE flag must not be set

Hostname specified by caller	AF	Hostname string contains	Result	Action
NULL hostname string; active	**AF_UNSPEC**	implied ::1 implied 127.0.0.1	one sockaddr_in6 and one sockaddr_in	inet_pton(AF_INET6) inet_pton(AF_INET)
	AF_INET6	implied ::1	one sockaddr_in6	inet_pton(AF_INET6)
	AF_INET	implied 127.0.0.1	one sockaddr_in	inet_pton(AF_INET)

Protocol-Independent Code

- Dealing with IPv4 and IPv6 at the same time complicates writing network programs: the code must be written to manage both types of addresses.
  - *protocol-dependent code*
- *getaddrinfo() and struct sockaddr_storage* introduce a first simplification of the problem
  - In a sense, they hide the IP version
- A better solution is to migrate the software to IPv6
  - relegating a few specific IPv4/IPv6 operations in a little number of library functions.
- Another help comes from *dual-stack* systems. They are designed to promote *IPv4/IPv6 Interoperability*
- What happens when clients and servers use different protocols?

IPv4 client - IPv6 server

- An *IPv4 client* **may** talk to an *IPv6 server* **if**:
  - The IPv6 server runs on a *dual-stack host*
  - The server has *both an A and an AAAA resource record* associated to its *hostname*
- The client:
  - asks the resolver for an A *resource record* and gets it
  - uses the server IPv4 address in connect()/sendto()

- The server runs on a dual-stack host:
  - the system will return to accept()/recvfrom() the client *IPv4-mapped-IPv6 address*
  - the server may use the **IN6_IS_ADDR_V4MAPPED()** macro to check whether the returned address is of the *IPv4-mapped-IPv6* type
  - the server may enable the **IPV6_V6ONLY** option to restrict connection to IPv6 clients only

IPv4 client - IPv6 server (da Stevens et al. *UNIX Network Programming* 3rd ed. ch.12)

We can summarize the steps that allow an IPv4 TCP client to communicate with an IPv6 server on a dual-stack host as follows:

1. The *IPv6* server starts, creates an *IPv6* listening socket, and it binds the wildcard address to the socket.

2. The *IPv4* client calls *getaddrinfo()* and finds an A record for the server. The server host will have both an A record and a AAAA record since it supports both protocols, but the IPv4 client asks for only an A record.

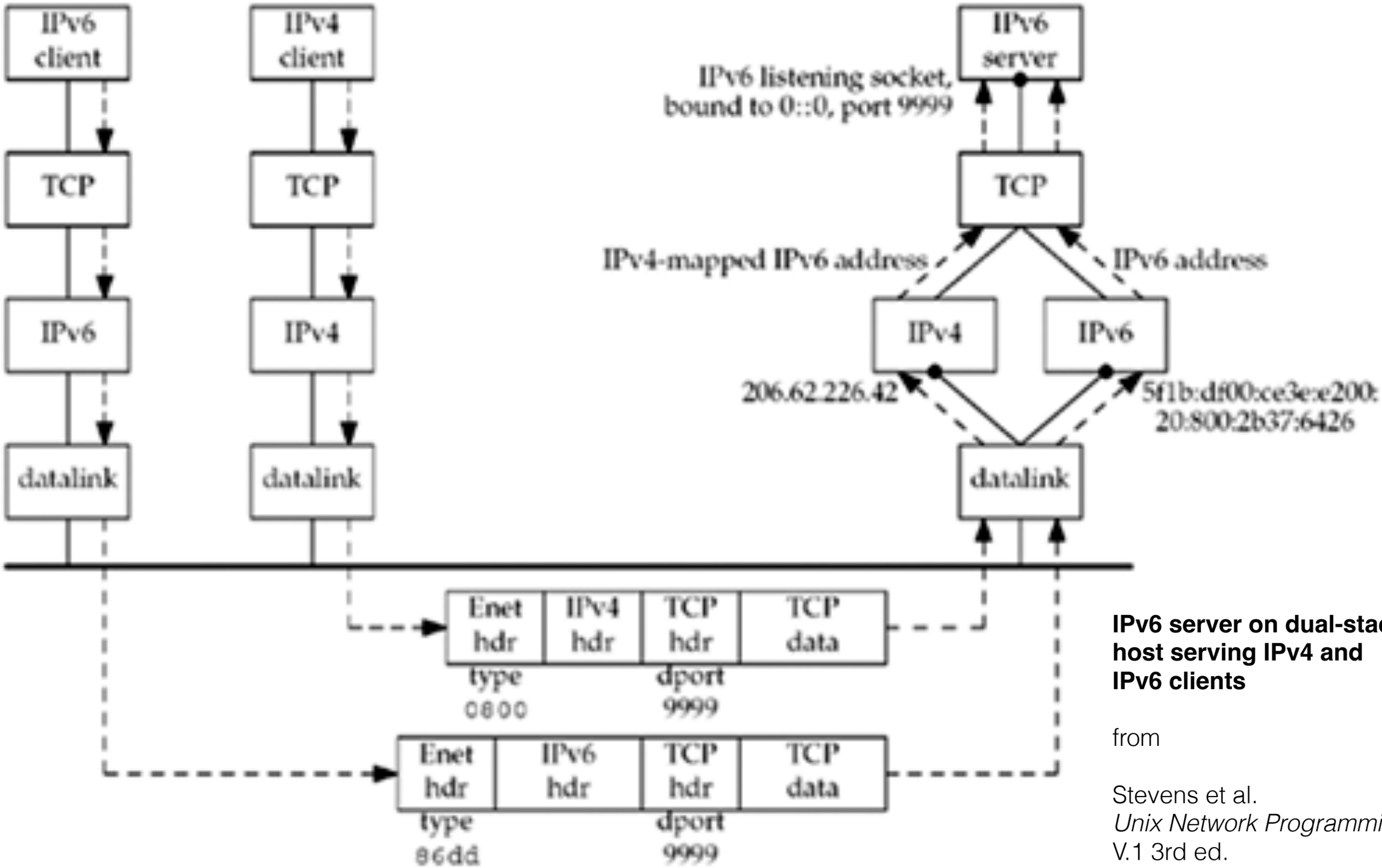
3. The client calls *connect()* and the client's host sends an *IPv4 SYN* to the server.

4. The server host receives the *IPv4 SYN* directed to the *IPv6* listening socket, sets a flag indicating that this connection is using *IPv4-mapped IPv6* addresses, and responds with an *IPv4 SYN/ACK*. When the connection is established, the address returned to the server by *accept()* is the *IPv4-mapped IPv6* address.

5. When the server host sends to the *IPv4-mapped IPv6* address, its IP stack generates *IPv4* datagrams to the *IPv4* address. Therefore, all communication between this client and server takes place using *IPv4* datagrams.

6. Unless the server explicitly checks whether this *IPv6* address is an *IPv4-mapped IPv6* address (using the *IN6_IS_ADDR_V4MAPPED* macro described in Section 12.4), the server never knows that it is communicating with an *IPv4* client. The dual-protocol stack handles this detail. Similarly, the *IPv4* client has no idea that it is communicating with an *IPv6* server.

IPv4 client - IPv6 server

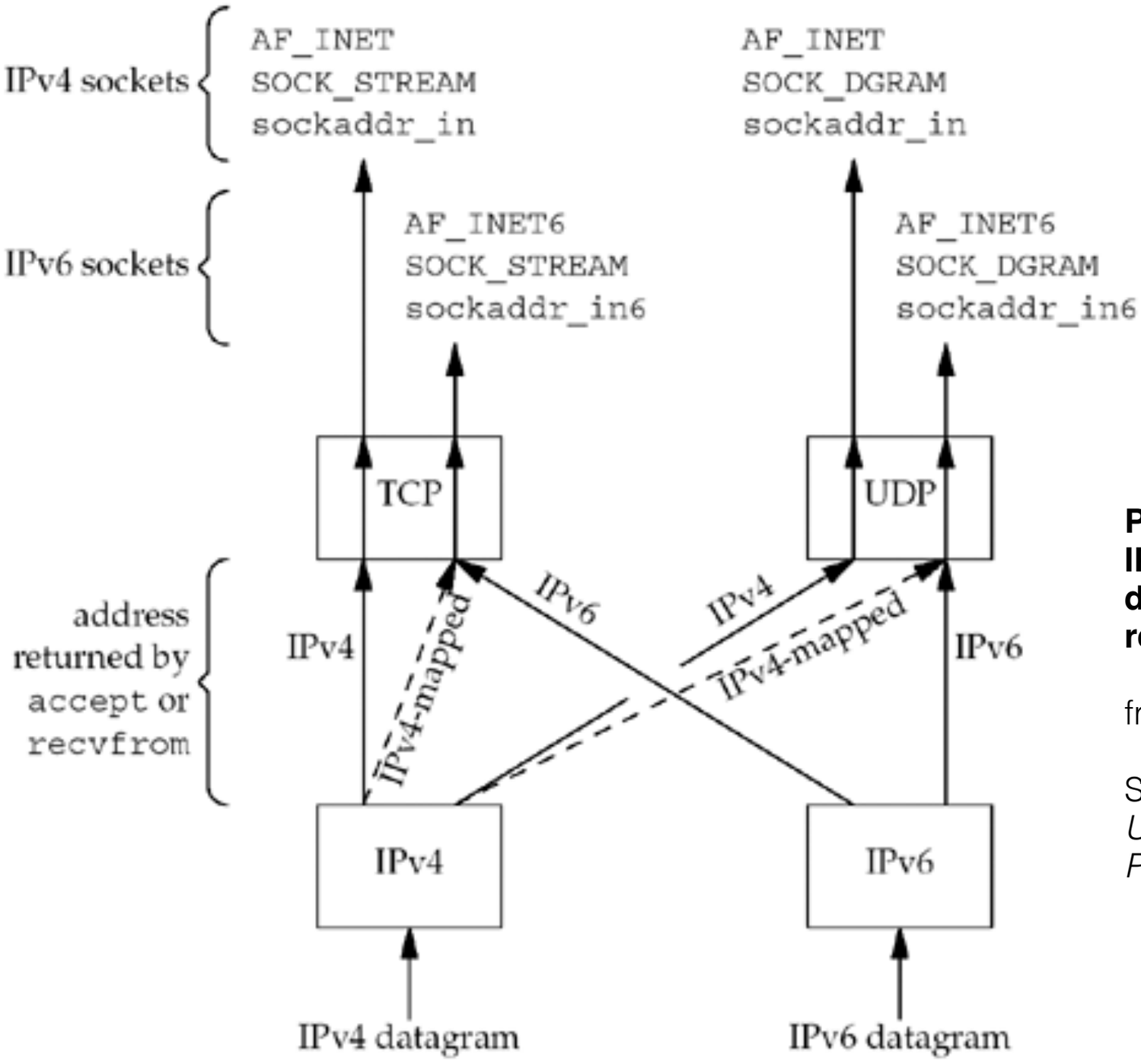


IPv6 server on dual-stack host serving IPv4 and IPv6 clients

from

Stevens et al.
Unix Network Programming
V.1 3rd ed.

IPv4 client - IPv6 server



Processing of received IPv4 or IPv6 datagrams, depending on type of receiving socket

from

Stevens et al.
Unix Network Programming V.1 3rd ed.

IPv6 client - IPv4 server

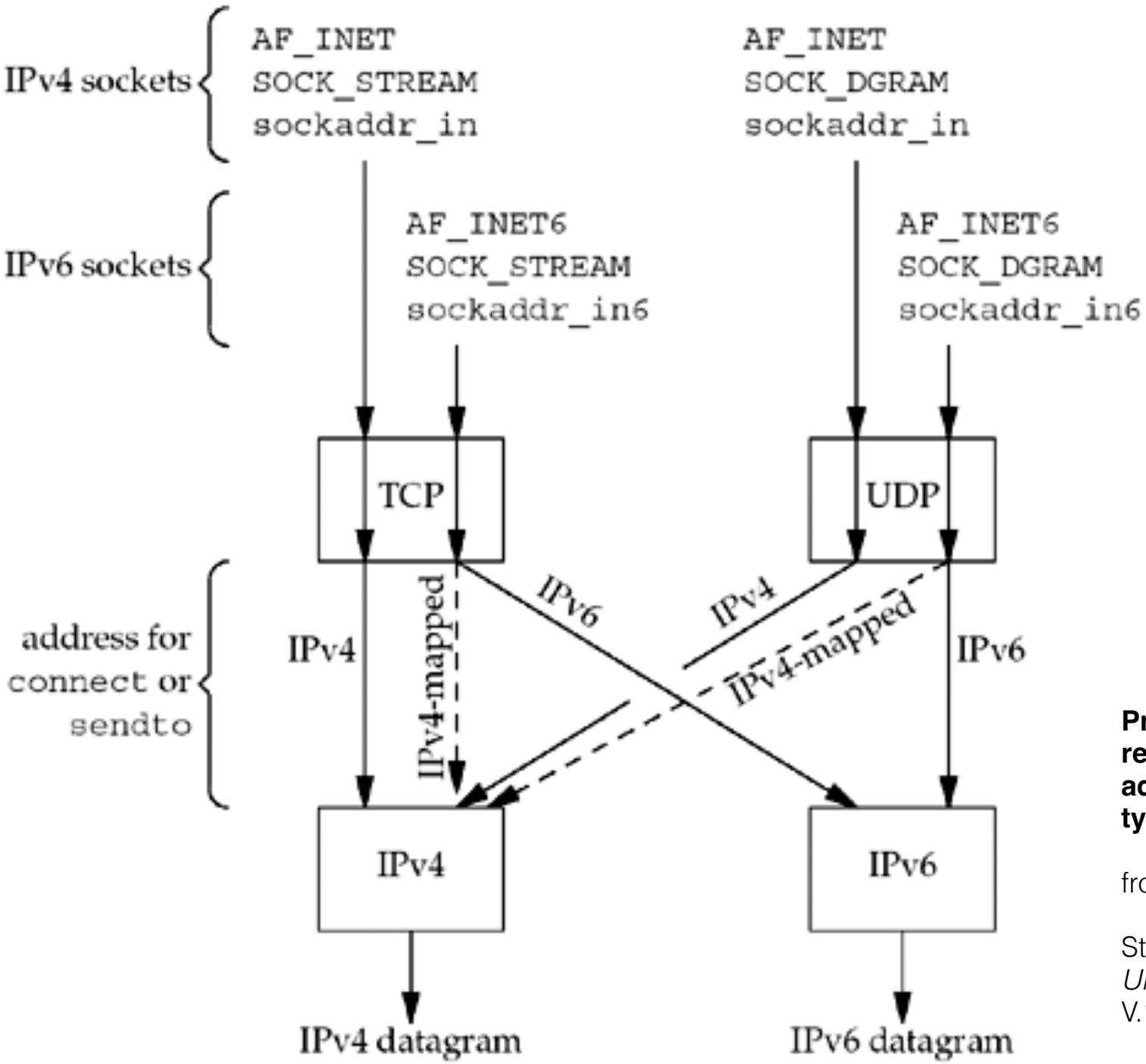
- An *IPv6 client* **may** talk to an *IPv4 server* **if**:
  - The IPv6 client runs on a *dual-stack host*
  - The client asks for the resolution with *AF_INET6* and the flag **AI_V4MAPPED** set
  - The resolver will return the server *IPv4-mapped-IPv6* address
  - The client will use such address (*sockaddr_in6*) in the connect()/sendto() calls directed to the server
- With connect(), an IPv4 connection will be established with the IPv4 server (on an IPv4 host)

IPv6 client - IPv4 server (da Stevens et al. *UNIX Network Programming* 3rd ed. ch.12)

We consider now an *IPv6* TCP client running on a dual-stack host:

1. An IPv4 server starts on an IPv4-only host and creates an IPv4 listening socket.
2. The IPv6 client starts and calls getaddrinfo() asking for only IPv6 addresses (it requests the AF_INET6 address family and sets the AI_V4MAPPED flag in its hints structure). Since the IPv4-only server host has only A records, we see from Figure 11.8 that an IPV4-mapped IPv6 address is returned to the client.
3. The IPv6 client calls connect() with the IPv4-mapped IPv6 address in the IPv6 socket address structure. The kernel detects the mapped address and automatically sends an IPv4 SYN to the server.
4. The server responds with an IPv4 SYN/ACK, and the connection is established using IPv4 datagrams.When the server host sends to the *IPv4-mapped IPv6* address, its IP stack generates *IPv4* datagrams to the *IPv4* address. Therefore, all communication between this client and server takes place using *IPv4* datagrams.

IPv6 client - IPv4 server



Processing of client requests, depending on address type and socket type

from

Stevens et al.
Unix Network Programming
V.1 3rd ed.

Protocol-Independent Code

- The following table [Stevens, ch. 12] summarizes the client-server interoperability options:

IPv4-IPv6 Interoperability	IPv4 server IPv4-only host (A only)	IPv6 server IPv6-only host (AAAA only)	IPv4 server dual-stack host (A + AAAA)	IPv6 server dual-stack host (A + AAAA)
IPv4 client IPv4-only host	IPv4	no	IPv4	IPv4
IPv6 client IPv6-only host	no	IPv6	no	IPv6
IPv4 client dual-stack host	IPv4	no	IPv4	IPv4
IPv6 client dual-stack host	IPv4	IPv6	no	IPv6

- When applications are run on a *dual-stack* host, communication is possible but:
  - IPv6 client on a dual-stack host and IPv4 server on a dual-stack host: if the client uses an address coming from an AAAA resource record, communication will not be possible
  - If it uses an IPv4-mapped-IPv6 coming from an A resource record, communication will be possible

IPv6 address-testing macro

- The SuSv4 standard defines in *netinet/in.h* some *macros* to reveal special *IPv6 addresses*:
  - They all take as input a ***(const struct in6_addr *address)*** and return a ***non null*** *value* if the IPv6 address ***is of the specified type***
    - *0* if it is not
    - For example, to test whether the address in (*struct in6_addr *ptr*) is an IPv4-mapped-IPv6 address, the following condition must be met:

IN6_IS_ADDR_V4MAPPED(ptr) != 0

```
#include <netinet/in.h>
IN6_IS_ADDR_UNSPECIFIED: Unspecified address
IN6_IS_ADDR_LOOPBACK: Loopback address
IN6_IS_ADDR_MULTICAST: Multicast address
IN6_IS_ADDR_LINKLOCAL: Unicast link-local address
IN6_IS_ADDR_SITELOCAL: Unicast site-local address
IN6_IS_ADDR_V4MAPPED: IPv4 mapped address
IN6_IS_ADDR_V4COMPAT: IPv4-compatible address

IN6_IS_ADDR_MC_NODELOCAL: Multicast node-local address.
IN6_IS_ADDR_MC_LINKLOCAL: Multicast link-local address.
IN6_IS_ADDR_MC_SITELOCAL: Multicast site-local address.
IN6_IS_ADDR_MC_ORGLOCAL: Multicast organization-local address.
IN6_IS_ADDR_MC_GLOBAL: Multicast global address.
```

I/O multiplexing

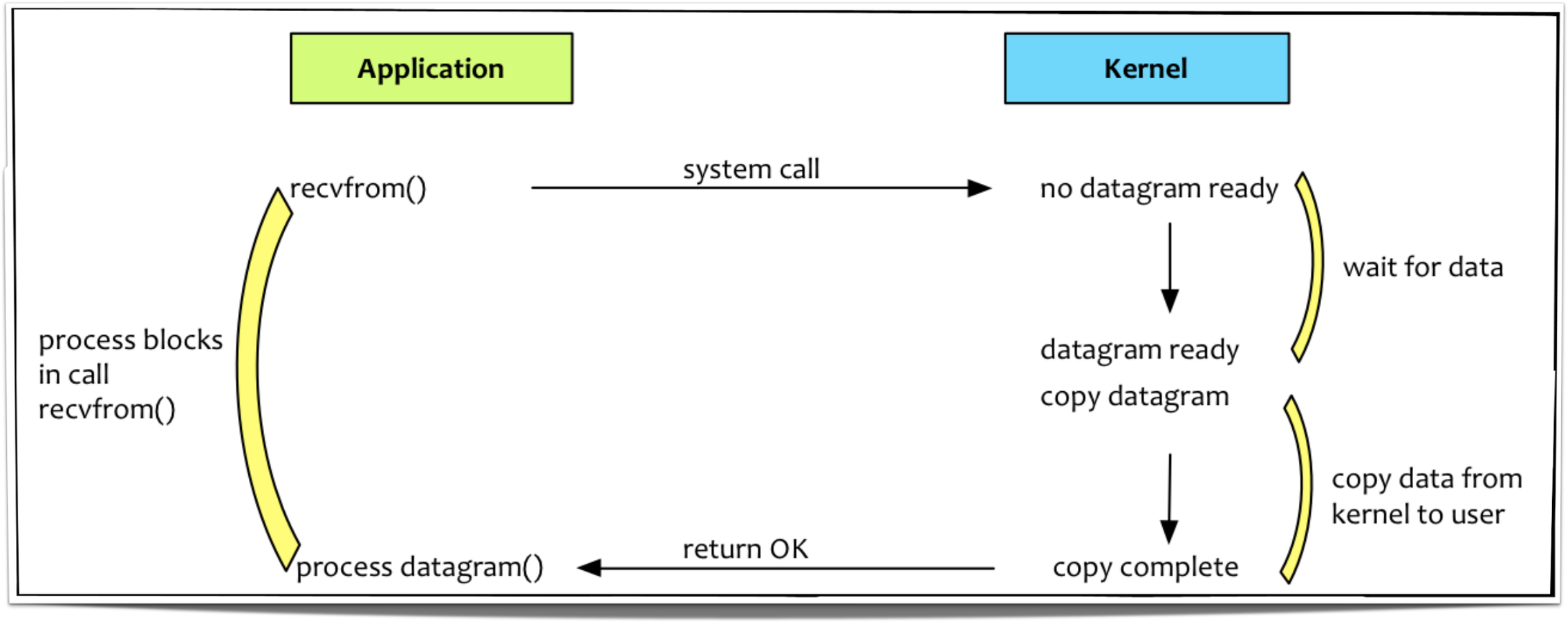
Catiuscia Melle and Franco Tommasi - SNP 2021-2022

I/O multiplexing

- Socket are **blocking** and so are system calls using them: calling process is blocked until the call execution is completed
- Sometime it is preferable for applications *not to block* (or at least *not to block for a single descriptor*) and to execute alternate code blocks
- This ability is called ***I/O multiplexing*** and it is provided by a few functions
    - *select()*, *pselect()*, *poll()*, ...
- Network applications use I/O multiplexing mainly **to manage more socket descriptors at the same time.** Some examples:
    - TCP sockets and UDP sockets, listening sockets (on different ports) and connected sockets, standard input file descriptor,...
    - A client application must manage more than 1 descriptor
        - the standard input (0) and 1 ore more socket descriptors
    - A TCP server application must manage at the same time both its *listening socket* and its *connected sockets*
        - *in general*, a server application managing more services and/or protocols (both TCP and UDP)

Blocking Mode: example

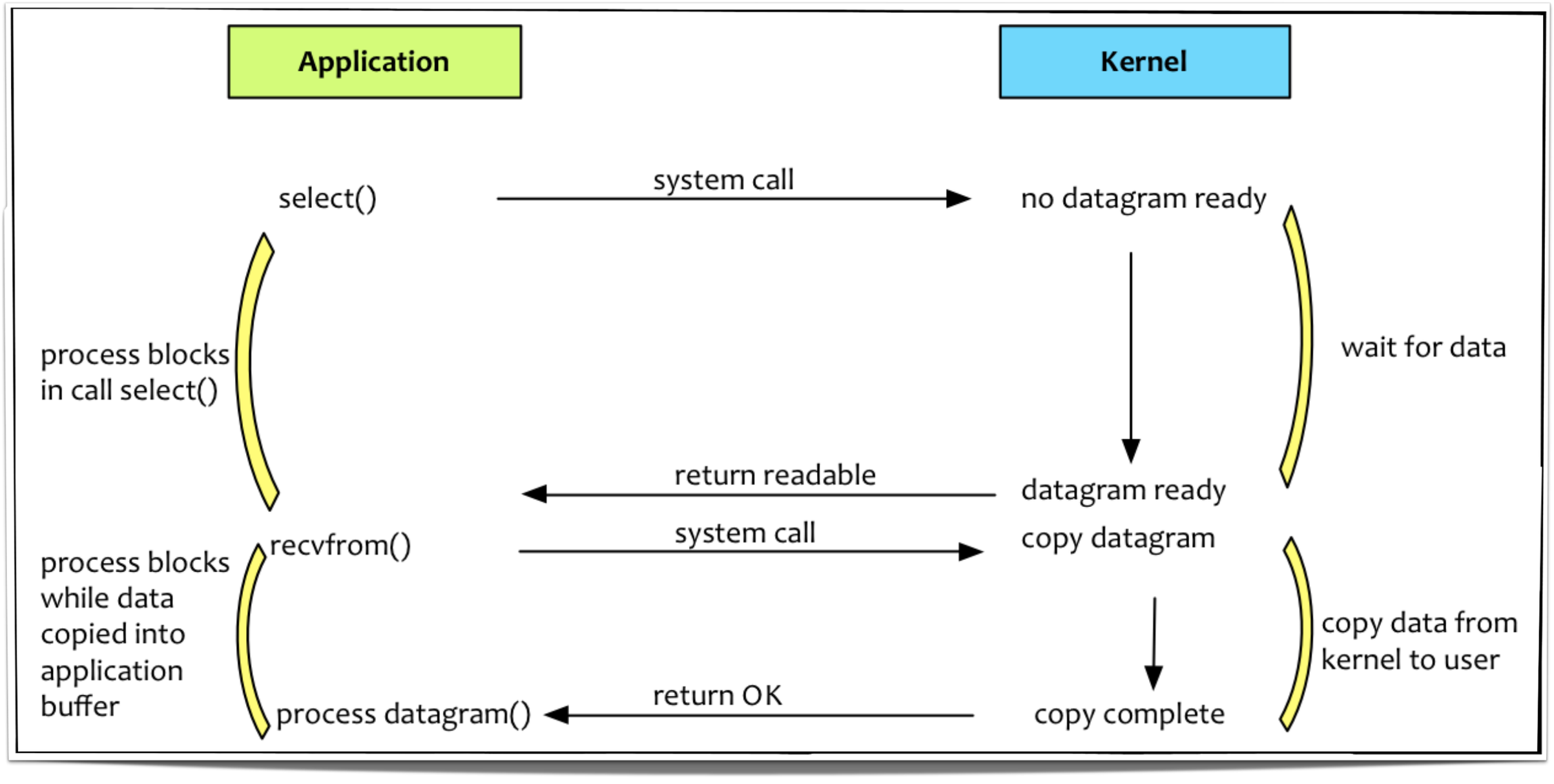
- An application calling *recvfrom()* will block until the reception of a message the kernel will copy in the application buffer.



- Only at this point, the process may continue its execution by processing the received datagram.

I/O Multiplexing: example

- The process is still blocked in the *select()* call, but this time has the advantage to be able to control **more** descriptors at the same time.



select()

- A process asks the kernel to wait for the conclusion of a set of events and to be reawakened when one (or some) of the events are concluded:
  - when one (or some) of the events are concluded *or*
  - after a given time interval has expired

```
#include <sys/select.h>
#include <sys/time.h>

int select(int maxfd1,
           fd_set *restrict readset,
           fd_set *restrict writeset,
           fd_set *restrict execptset,
           struct timeval *restrict timeout);
```

- The return value is an integer representing:
  - **-1** in case of error
  - **0** after timeout expiration
  - **>0** the number of ready descriptors (in all sets)

select(): struct timeval

- The *struct timeval* allows to specify a waiting time (that is *select() blocking*) in *seconds* and *microseconds*:

```
struct timeval {
  long tv_sec; //seconds
  long tv_usec; //microseconds
};
```

  - **timeout == NULL**, *select()* waits *forever* until *at least 1* of the specified descriptors is ready for I/O;
  - **timeout == finite value**, this is the maximum time a process is willing to wait the I/O to complete;
  - **timeout == 0** (*tv_sec=tv_usec=0*), *select()* returns immediately after checking the descriptor sets (*polling).*

select()

- **maxfd1** *(0 <* **maxfd1** *< FD_SETSIZE)* specifies ***how many*** descriptors to monitor:
  - the kernel will test the first **maxfd1** descriptors in each set, from 0 to (**maxfd1**-*1)*.
    - if the highest descriptor is *5*, the value of **maxfd1** will be *6*, to tell the kernel it must control descriptors from 0 to 5.
- If ***readset*** is not NULL, it points to an object of type *fd_set* that indicates:
  - in input: *which* descriptors to monitor for their readiness to be read
  - in output: *which* of them are ready to be read;
- If ***writeset*** is not NULL, it points to an object of type *fd_set* that indicates:
  - in input: *which* descriptors to monitor for their readiness to be written
  - in output: *which* of them are ready to be written;
- If ***execptset*** is not NULL, it points to an object of type *fd_set* that indicates:
  - in input: *which* descriptors to monitor for exception conditions
  - in output: *which* of them have exception conditions pending

select() and the file descriptors sets

- The kernel will keep *readset*, *writeset* ed *execptset* under control and will return from *select()* when *at least* one of the monitored descriptors is ready for *reading*, *writing* or for *exceptions*
  - this implies *select()* ***modifies*** *the objects pointed by readset, writeset ed execptset*: on return they are overwritten to indicate *which* descriptors are ***ready***
  - the return value of *select()* - **if greater than 0** - gives *the total number of descriptors ready* in *all* output sets
- *file descriptor set* management is performed by the following 4 macros
  - or functions, (implementation-dependent)
  - they operate on *file descriptor masks* of type **fd_set**:

```
void FD_ZERO(fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
int  FD_ISSET(int fd, fd_set *fdset);
```

- **FD_ZERO** zeroes **fdset**;
- **FD_SET** *enables the bit corresponding to fd* in *fdset*
- **FD_CLR** dis*ables the bit corresponding to fd* in *fdset*
- **FD_ISSET** *verifies* whether fd is *on* in fdset:
  - **returns a value ≠ 0 if fd is in fdset, 0 otherwise**

Ready conditions

- A descriptor is considered **ready for reading** when enough data are available for the process to read them from that descriptor without blocking. This may happen when:
  - the number of bytes available in the socket receive buffer is **greater** than the ***receive low-water mark*** defined for the socket
    - it defaults to 1 for TCP/UDP
    - a *recv/recvfrom* call *will not* block and will return a value greater than 0

- In particular, a read operation will not block and will return:
  - number of read bytes >= 0
    - number of read bytes = 0 in case of a closing TCP connection
  - number of read bytes = -1 (pending error condition di on the socket)
    - **errno** is set to the occurred error
      - the error can be read by *getsockopt()* and the *SO_ERROR* option
  - for a listening socket, *ready* means there is at list a connection ready to be *accept()*ed
    - in this case, an *accept()* will not block and will return the connected socket

Ready conditions

- A descriptor is considered **ready for writing** when a writing operation on the descriptor may be performed without blocking the process. This may happen when:
  - the number of free bytes in the socket send buffer is **greater** than the defined *send low-water mark* value and the socket is connected or the socket does not require a connection
    - the default “send low water mark” is 2048 for TCP/UDP
  - *A call send/sendto will not block and will return number of bytes accepted by the transport layer*
  - the socket has been **closed in writing**
    - if the connection has been closed in writing, a further writing will return an *EPIPE* error and a *SIGPIPE* will be generated for the calling process
  - a socket in **nonblocking** mode concluded a *connect()*
    - or connect() failed
  - there is a **pending error** for the socket:
    - pending error for the socket: a write to the socket will be nonblocking, and will return **-1** with **errno** set to indicate the occurred error

Socket options

How to modify the sockets' default behavior

Catiuscia Melle and Franco Tommasi - SNP 2021-2022

Socket options

- Each socket has **options** which define its **behaviour.** Such options have **default values.**

- To functions are available to manage *socket options*:
  - *setsockopt()*: sets socket options for a socket (i.e. modifies their default values)
  - *getsockopt()*: read the current value of a socket option for a socket

```
#include <sys/socket.h>
                    int setsockopt(int socket, int level, int option_name,
                        const void *option_value, socklen_t option_len);

                    int getsockopt(int socket, int level, int option_name,
                  void *restrict option_value, socklen_t *restrict option_len);
```

- Both:
  - Take in input the file descriptor of the socket of interest
  - Both return:
    - **0** in case of success
    - **-1** in case of error, setting the value of *errno*

Socket options

- In both calls:
  - ***socket*** must be an *open socket descriptor*
  - ***level*** specifies whether the option is *general* or *protocol-specific*
  - ***option_name***, ***option_value***, ***option_len*** define:
    - on *which option they* act,
    - *which value* to assign to it
    - *which is the size* of the *option_value* variable
  - *option_value* is the value of *option_name* to be **set** by **setsockopt()**,
  - *option_value* is where the value of *option_name* read by **getsockopt()** is **stored**
    - in this case, also *option_len* is a value-result argument
  - options may be ***binary*** (when they enable/disable some features or *flags*) or they may ***assign specific values***

Socket options

- Socket options are selected by a couple of values: (***level, optname***)
  - *Level* indicates the protocol on which to operate (transport, network)
    - each level has its own *set of specific options*
  - *Optname* is the name of the option
    - the option must be supported by the chosen level
- Possible values for *level* are:
  - ***SOL_SOCKET*** (as defined in *sys/socket.h*), deal with generic options for sockets
  - or specific for a protocol (as defined in *netinet/in.h*):

    `IPPROTO_IP, IPPROTO_IPV6, IPPROTO_TCP, ...`

Socket options

- Once the pair (*level*, *optname*) is selected:
- for **setsockopt()**, the **value** (*option_value)* and its **size** (*option_len)*, must be provided
  - *option_value* is usually an integer: a 0 disables the option, a different value activates it
  - if the option doesn't need a value, then *option_value* will be 0.
- for **getsockopt()**, *option_value* will receive the read value

Main options

- Among the generic options for sockets, found in ***sys/socket.h***, we find:

SO_ACCEPTCONN: Socket is accepting connections
SO_BROADCAST: Transmission of broadcast messages is supported
SO_DEBUG: Debugging information is being recorded
SO_DONTROUTE: Bypass normal routing
SO_ERROR: Socket error status
SO_KEEPALIVE: Connections are kept alive with periodic messages
SO_LINGER: Socket lingers on close
SO_OOBINLINE: Out-of-band data is transmitted in line
SO_RCVBUF: Receive buffer size
SO_RCVLOWAT: Receive "low water mark"
SO_RCVTIMEO: Receive timeout
SO_REUSEADDR: Reuse of local addresses is supported
SO_SNDBUF: Send buffer size
SO_SNDLOWAT: Send "low water mark"
SO_SNDTIMEO: Send timeout
SO_TYPE: Socket type

Main options

- The type of the options is recapitulated in
  - http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_10_16

Table: Socket-Level Options

Option	Parameter Type	Parameter Meaning
SO_ACCEPTCONN	**int**	Non-zero indicates that socket listening is enabled (*getsockopt()* only).
SO_BROADCAST	**int**	Non-zero requests permission to transmit broadcast datagrams (SOCK_DGRAM sockets only).
SO_DEBUG	**int**	Non-zero requests debugging in underlying protocol modules.
SO_DONTROUTE	**int**	Non-zero requests bypass of normal routing; route based on destination address only.
SO_ERROR	**int**	Requests and clears pending error information on the socket (*getsockopt()* only).
SO_KEEPALIVE	**int**	Non-zero requests periodic transmission of keepalive messages (protocol-specific).
SO_LINGER	**struct linger**	Specify actions to be taken for queued, unsent data on *close()*: linger on/off and linger time in seconds.
SO_OOBINLINE	**int**	Non-zero requests that out-of-band data be placed into normal data input queue as received.
SO_RCVBUF	**int**	Size of receive buffer (in bytes).
SO_RCVLOWAT	**int**	Minimum amount of data to return to application for input operations (in bytes).
SO_RCVTIMEO	**struct timeval**	Timeout value for a socket receive operation.
SO_REUSEADDR	**int**	Non-zero requests reuse of local addresses in *bind()* (protocol-specific).
SO_SNDBUF	**int**	Size of send buffer (in bytes).
SO_SNDLOWAT	**int**	Minimum amount of data to send for output operations (in bytes).
SO_SNDTIMEO	**struct timeval**	Timeout value for a socket send operation.
SO_TYPE	**int**	Identify socket type (*getsockopt()* only).

Options-1

level	*optname*	get	set	Description	Flag	Datatype
SOL_SOCKET	SO_BROADCAST	•	•	Permit sending of broadcast datagrams	•	int
	SO_DEBUG	•	•	Enable debug tracing	•	int
	SO_DONTROUTE	•	•	Bypass routing table lookup	•	int
	SO_ERROR	•		Get pending error and clear		int
	SO_KEEPALIVE	•	•	Periodically test if connection still alive	•	int
	SO_LINGER	•	•	Linger on close if data to send		linger{}
	SO_OOBINLINE	•	•	Leave received out-of-band data inline	•	int
	SO_RCVBUF	•	•	Receive buffer size		int
	SO_SNDBUF	•	•	Send buffer size		int
	SO_RCVLOWAT	•	•	Receive buffer low-water mark		int
	SO_SNDLOWAT	•	•	Send buffer low-water mark		int
	SO_RCVTIMEO	•	•	Receive timeout		timeval{}
	SO_SNDTIMEO	•	•	Send timeout		timeval{}
	SO_REUSEADDR	•	•	Allow local address reuse	•	int
	SO_REUSEPORT	•	•	Allow local port reuse	•	int
	SO_TYPE	•		Get socket type		int
	SO_USELOOPBACK	•	•	Routing socket gets copy of what it sends	•	int
IPPROTO_IP	IP_HDRINCL	•	•	IP header included with data	•	int
	IP_OPTIONS	•	•	IP header options		(see text)
	IP_RECVDSTADDR	•	•	Return destination IP address	•	int
	IP_RECVIF	•	•	Return received interface index	•	int
	IP_TOS	•	•	Type-of-service and precedence		int
	IP_TTL	•	•	TTL		int
	IP_MULTICAST_IF	•	•	Specify outgoing interface		in_addr{}
	IP_MULTICAST_TTL	•	•	Specify outgoing TTL		u_char
	IP_MULTICAST_LOOP	•	•	Specify loopback		u_char
	IP_{ADD,DROP}_MEMBERSHIP		•	Join or leave multicast group		ip_mreq{}
	IP_{BLOCK,UNBLOCK}_SOURCE		•	Block or unblock multicast source		ip_mreq_source{}
	IP_{ADD,DROP}_SOURCE_MEMBERSHIP		•	Join or leave source-specific multicast		ip_mreq_source{}
IPPROTO_ICMPV6	ICMP6_FILTER	•	•	Specify ICMPv6 message types to pass		icmp6_filter{}
IPPROTO_IPV6	IPV6_CHECKSUM	•	•	Offset of checksum field for raw sockets		int
	IPV6_DONTFRAG	•	•	Drop instead of fragment large packets	•	int
	IPV6_NEXTHOP	•	•	Specify next-hop address		sockaddr_in6{}
	IPV6_PATHMTU	•		Retrieve current path MTU		ip6_mtuinfo{}
	IPV6_RECVDSTOPTS	•	•	Receive destination options	•	int
	IPV6_RECVHOPLIMIT	•	•	Receive unicast hop limit	•	int
	IPV6_RECVHOPOPTS	•	•	Receive hop-by-hop options	•	int
	IPV6_RECVPATHMTU	•	•	Receive path MTU	•	int
	IPV6_RECVPKTINFO	•	•	Receive packet information	•	int
	IPV6_RECVRTHDR	•	•	Receive source route	•	int
	IPV6_RECVTCLASS	•	•	Receive traffic class	•	int
	IPV6_UNICAST_HOPS	•	•	Default unicast hop limit		int
	IPV6_USE_MIN_MTU	•	•	Use minimum MTU	•	int
	IPV6_V6ONLY	•	•	Disable v4 compatibility	•	int
	IPV6_*XXX*	•	•	Sticky ancillary data		(see text)
	IPV6_MULTICAST_IF	•	•	Specify outgoing interface		u_int
	IPV6_MULTICAST_HOPS	•	•	Specify outgoing hop limit		int
	IPV6_MULTICAST_LOOP	•	•	Specify loopback	•	u_int
	IPV6_JOIN_GROUP		•	Join multicast group		ipv6_mreq{}
	IPV6_LEAVE_GROUP		•	Leave multicast group		ipv6_mreq{}
IPPROTO_IP *or* **IPPROTO_IPV6**	MCAST_JOIN_GROUP		•	Join multicast group		group_req{}
	MCAST_LEAVE_GROUP		•	Leave multicast group		group_source_req{}
	MCAST_BLOCK_SOURCE		•	Block multicast source		group_source_req{}
	MCAST_UNBLOCK_SOURCE		•	Unblock multicast source		group_source_req{}
	MCAST_JOIN_SOURCE_GROUP		•	Join source-specific multicast		group_source_req{}
	MCAST_LEAVE_SOURCE_GROUP		•	Leave source-specific multicast		group_source_req{}

Options-2

level	*optname*	get	set	Description	Flag	Datatype
IPPROTO_TCP	TCP_MAXSEG	•	•	TCP maximum segment size		int
	TCP_NODELAY	•	•	Disable Nagle algorithm	•	int
IPPROTO_SCTP	SCTP_ADAPTION_LAYER	•	•	Adaption layer indication		sctp_setadaption{}
	SCTP_ASSOCINFO	†	•	Examine and set association info		sctp_assocparams{}
	SCTP_AUTOCLOSE	•	•	Autoclose operation		int
	SCTP_DEFAULT_SEND_PARAM	•	•	Default send parameters		sctp_sndrcvinfo{}
	SCTP_DISABLE_FRAGMENTS	•	•	SCTP fragmentation	•	int
	SCTP_EVENTS	•	•	Notification events of interest		sctp_event_subscribe{}
	SCTP_GET_PEER_ADDR_INFO	†		Retrieve peer address status		sctp_paddrinfo{}
	SCTP_I_WANT_MAPPED_V4_ADDR	•	•	Mapped v4 addresses	•	int
	SCTP_INITMSG	•	•	Default INIT parameters		sctp_initmsg{}
	SCTP_MAXBURST	•	•	Maximum burst size		int
	SCTP_MAXSEG	•	•	Maximum fragmentation size		int
	SCTP_NODELAY	•	•	Disable Nagle algorithm	•	int
	SCTP_PEER_ADDR_PARAMS	†	•	Peer address parameters		sctp_paddrparams{}
	SCTP_PRIMARY_ADDR	†	•	Primary destination address		sctp_setprim{}
	SCTP_RTOINFO	†	•	RTO information		sctp_rtoinfo{}
	SCTP_SET_PEER_PRIMARY_ADDR		•	Peer primary destination address		sctp_setpeerprim{}
	SCTP_STATUS	†		Get association status		sctp_status{}

SO_TYPE

- SO_TYPE allows to read the *type* of a socket
  - SOCK_STREAM, SOCK_DGRAM
  - when a process inherits a socket from a parent, it can get its type by *getsockopt()*:

```
int socktype;
socklen_t optlen = sizeof(socktype);

int res = getsockopt(sockfd, SOL_SOCKET, SO_TYPE, &socktype, &optlen);
```

SO_REUSEADDR

- *SO_REUSEADDR*: allows to perform a *bind()* on a socket even if the specified address is used by another socket.
  - if this option is not set, *bind()* returns the *EADDRINUSE* error

```
int optval = 1;
socklen_t optlen = sizeof(optval);
int res =
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &optval, optlen);
```

  - Scenario 1, TCP server:
    - a listening server is started and, when a connection arrives, the client is handed to a child process;
    - the parent terminates while the child is still executing.
    - If the parent is restarted it will not be able to perform a bind() to the well-known port since there is already a socket using it (in the child) forbidding the parent's new instance to be activated on the same address: with the SO_REUSEADDR option, *bind()* will succeed
- Scenario 2, UDP server: the option allows a UDP server to perform a *complete duplicate binding* of the same IP and the same port on more than one socket
  - *multicast application*

SO_ERROR

- If an *error* occurs on a socket, the kernel stores in the **so_error** variable the *error pending* for the *socket*
  - With the *Exxx value* (from the Unix standard) representing it
- The process may use the **SO_ERROR** option to read the value of *so_error*:

```
int errorValue;
socklen_t optlen = sizeof(errorValue);
int res = getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &errorValue, &optlen);
```

  - *getsockopt()* returns, in the interger variable pointed by *errorValue*, the pending error code and **so_error** is reset to **0** by the kernel
- It is a read-only option
  - No *setsockopt()* for it

SO_ERROR

- If a process *reads* from a socket with a pending error and *there are no data in the receive buffer*:
  - *recv()* returns **-1**,
  - *errno* is set to *so_error*,
  - *so_error = 0* by the kernel.
- If there are data in the receive buffer, the message is returned and not the error
- If a process *writes* to a socket with a pending error:
  - *send()* returns **-1**,
  - *errno = so_error*
  - *so_error = 0* by the kernel
- If a process is blocked in a select(), writing and/or reading from a socket, select() returns the socket ready both in writing and in reading

SO_BROADCAST

- *SO_BROADCAST*: enable/disable sending broadcast messages for a process
  - make sense only for UDP socket
  - and for networks supporting broadcast transmission
    - e.g. Ethernet, but not point-to-point links
- If a process tries to send broadcast datagrams without the rights to do so, an *EACCESS* error will be generated

```
int optval = 1;
socklen_t optlen = sizeof(optval);
int res =
   setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &optval, optlen);
```

```
int optval;
socklen_t optlen = sizeof(optval);
int res =
   getsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &optval, &optlen);
```

SO_KEEPALIVE

- *SO_KEEPALIVE* asks the kernel to send *keep-alive probes* to verify whether the connection is still active after 2 hours of inactivity (TCP)
  - useful to test network failures or silent crashes of the remote node (preventing its closing of the socket descriptor)
- TCP sends the probe and waits for an answer:
  - If the peer sends back an ACK, the timer is reset - and a new probe will be sent after 2 hours of inactivity;
  - If the peer sends back a *RST*, TCP infers the peer crashed and rebooted: pending error for the socket is set to ***ECONNRESET*** and the socket is closed;
  - With no answer after 8 new probes, at 75 seconds intervals, pending error for the socket is set to ***ETIMEDOUT*** and the socket is closed

SO_KEEPALIVE

- Option SO_KEEPALIVE is inherited by a socket coming from a listening socket
  - it is active for a socket returned by *accept()* - if it was set for the *listening socket*
- The option has been introduced to detect peers' crashes or host unreachable situations
  - Beware: with this option active, network faults may cause the termination of perfectly valid TCP connections.

- An application level keep-alive mechanism offers better features:
  - the process is the only responsible for its own TCP connections management
  - the process has means to evaluate the network state and can more effectively decide what is better for the application's users
  - the process may decide to free connection resources when it realizes they are not needed anymore

SO_KEEPALIVE

- TCP connection possible conditions

Scenario	Peer process crashes	Peer host crashes	Peer host is unreachable
local TCP actively sending data	Peer TCP sends a **FIN** (may be immediately intercepted by a *select()* waiting to read from the socket): if the local TCP sends another segment, the peer answer with a **RST**. A further writing attempt will generate a **SIGPIPE** for the local socket TCP	Peer TCP does not sends a **FIN:** local TCP will timeout and the socket will return the **ETIMEDOUT** pending error	Local TCP will timeout and the socket will return the **EHOSTUNREACH** pending error
local TCP actively receiving data	Peer TCP sends a **FIN** which will be understood as an **EOF** (premature)	Local TCP terminates data reception	Local TCP terminates data reception
connection is idle, keep-alive set	Peer TCP sends a **FIN** (may be immediately intercepted by a *select()* waiting to read from the socket)	9 keep-alive probes are sent after 2 hrs of inactivity, then the socket will return the **ETIMEDOUT** pending error	9 keep-alive probes are sent after 2 hrs of inactivity, then the socket will return the **EHOSTUNREACH** pending error
connection is idle, keep-alive not set	Peer TCP sends a **FIN** (may be immediately intercepted by a *select()* waiting to read from the socket)	nothing	nothing

SO_LINGER

- SO_LINGER option modifies *close()'s behaviour* for a *TCP socket*
  - *close()* returns immediately control to the process, leaving to the kernel the management of the TCP connection closing and the sending of data possibly still present in the *send buffer*
- When the SO_LINGER option is active, close() becomes **blocking**
  - until the closing process is completed,
  - or until expiration of the timeout set by the option.
- The option uses a ***struct linger*** :

```
#include <sys/socket.h>

struct linger {
 int l_onoff;   /* if ≠ 0, activates the option */
 int l_linger;  /* if ≠ 0, tells how many seconds
        close() blocks before returning */
};
```

SO_LINGER

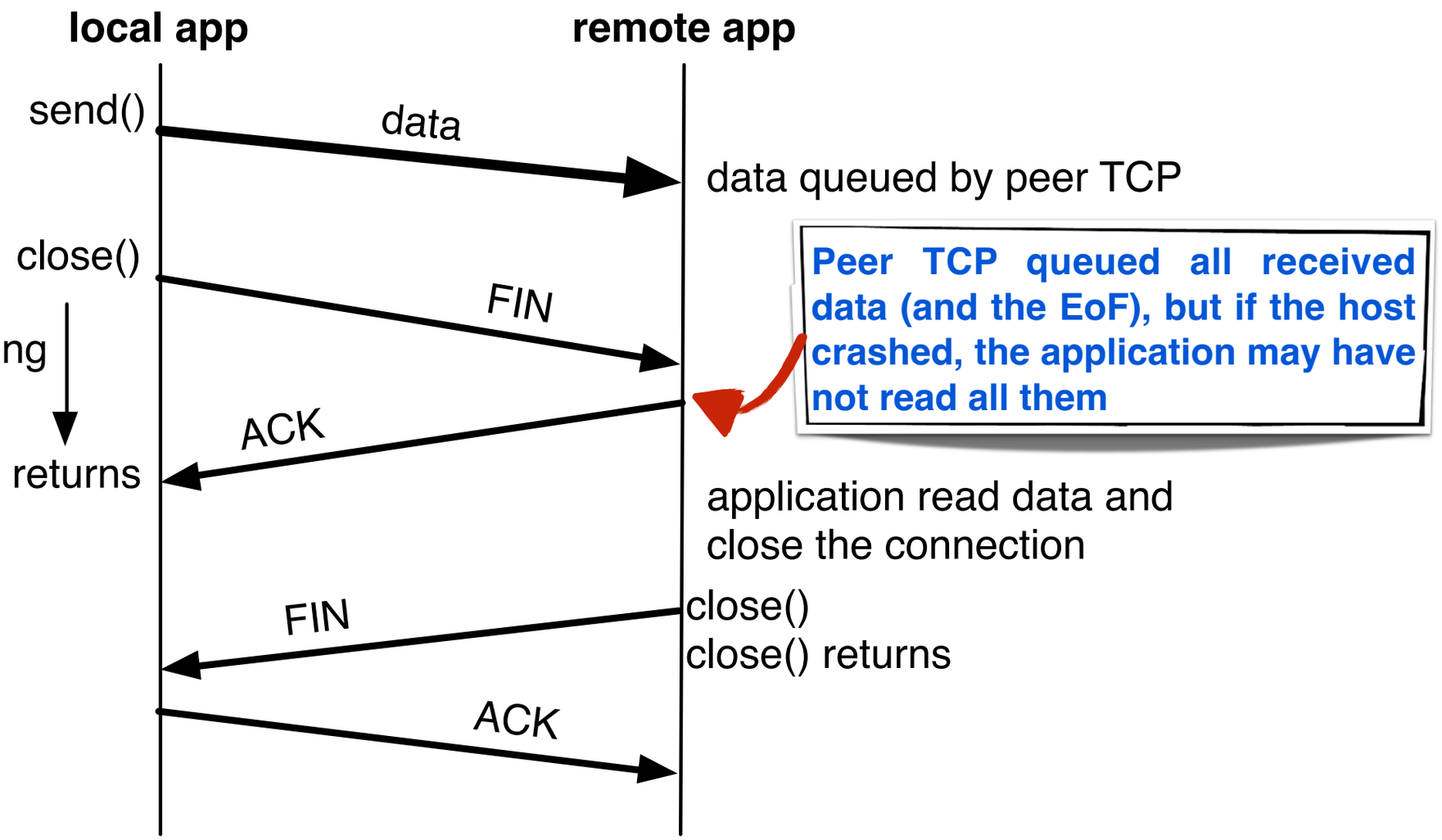
- In *struct linger*, if **l_onoff = 0** ⇒ *the option is deactivated*
  - *l_linger* value is ignored, *close()* returns immediately
    - default behaviour
- if **l_onoff ≠ 0 =>** *the option is active*
  - close() behaviour depends on *l_linger* value:
    - **l_linger = 0**: TCP performs a connection **abort** when *close()* is invoked
      - TCP **discards** data still in the send buffer and sends an **RST** to the peer
      - Thus avoiding the *TIME_WAIT* state for the socket
    - **l_linger ≠ 0**: when close() is called, the process blocks:
      - until TCP sends **all** data still in the send buffer, waiting for the peer's ACK for all data and for the closing FIN
      - When *linger time* expires: close() will return an **EWOULDBLOCK** error and data still in send buffer will be discarded.

```
struct linger lopt;
lopt.l_onoff = 1;
lopt.l_onoff = 1;

setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &lopt, sizeof(lopt));
```

Closing a TCP connection: linger

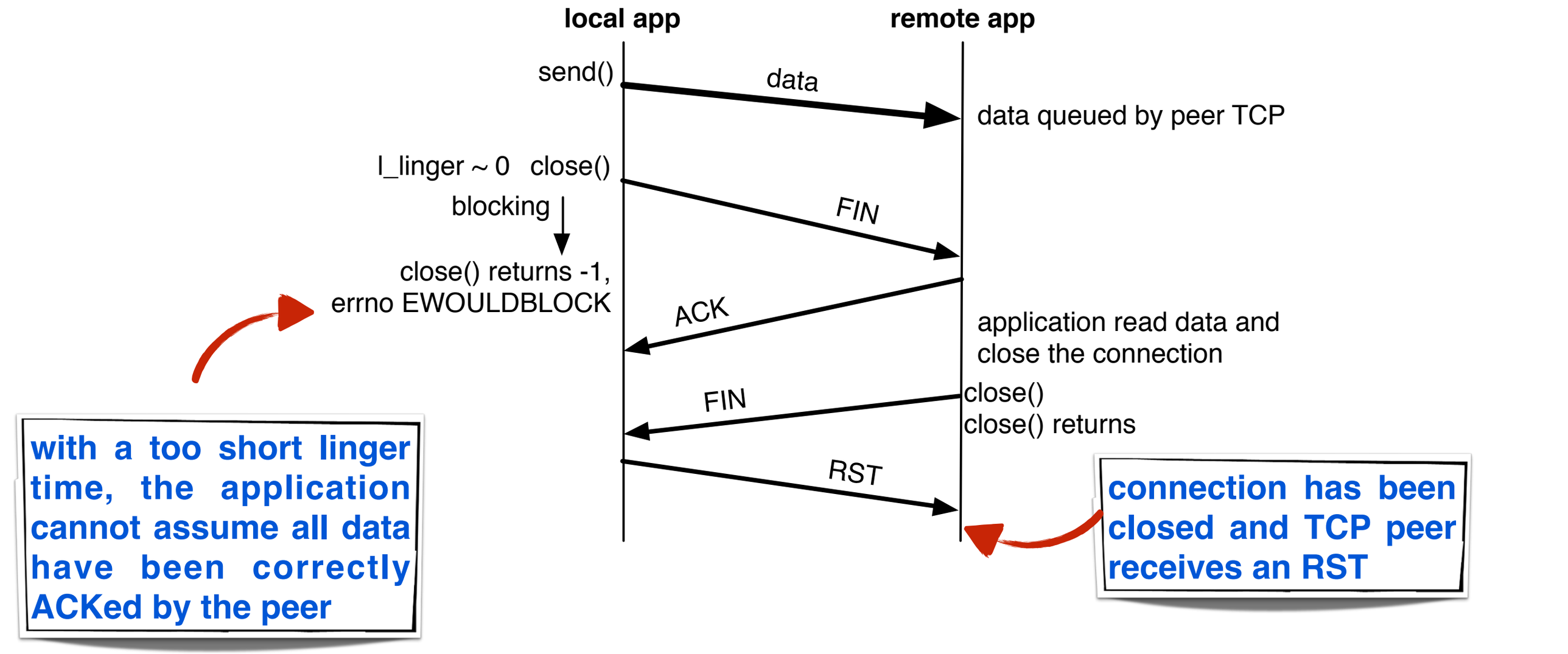
- If the application enables the **SO_LINGER** option for a TCP socket, *close()* will be blocking
  - If *l_linger > 0*, until all data (and the FIN) have been ACKed from remote peer, close() will block

At this point the application knows sent data reached the remote TCP and that the sent FIN has been ACKed. However it doesn't know whether the remote application received them.



Closing a TCP connection: linger

- If the **SO_LINGER** option is active for a TCP socket with a too short linger time, the application cannot be assured the data transfer has been successfully completed



Closing a TCP connection

- A comparison among *close()*, **half-close** (*shutdown()*) and use of *SO_LINGER*

Function	Description
shutdown, SHUT_RD	The socket is closed for reading: no more recv() possible, send() possible. Data still in *RECV Queue* are discarded, further received data discarded
shutdown, SHUT_WR (half-close)	The socket is closed for writing: no more send() possible, recv() possible. Data still in SEND Queue are transmitted and ACKed, then the closing FIN is sent
close, l_onoff = 0 (default)	The socket is closed for reading and for writing: no more recv() or send() possible. Data still in SEND Queue are transmitted to the peer. If the socket reference count is 0, the 4-way teardown starts: data still in *RECV Queue* are discarded and a FIN is queued in SEND Queue
close, l_onoff = 1, l_linger=0	The socket is closed for reading and for writing. If the socket reference count is 0, than a RST is sent to the peer TCP and the socket enters the CLOSED state (without doing a TIME_WAIT). The content of SEND Queue and RECV Queue is discarded.
close, l_onoff = 1, l_linger≠0	The socket is closed for reading and for writing. Data still in SEND Queue are transmitted to the peer. If the socket reference count is 0, the 4-way teardown starts: FIN is sent after all data and the content of RECV Queue is discarded. If linger time expires before the operations' end, close() returns with -1 and errno=EWOULDBLOCK

Socket I/O TIMEOUT

- **SO_RCVTIMEO** and **SO_SNDTIMEO** options allow to set a *timeout* for read and write operation with a socket
  - SO_RCVTIMEO: *recv(), recvfrom()*
  - SO_SNDTIMEO: *send(), sendto()*
- *setsockopt()* e *getsockopt()* use a *struct timeval* to set or return the timeout value for the socket
  - same struct used in in *select()* (timeout in *seconds* and *microseconds*)

```
struct timeval time;
time.tv_sec = 5;
time.tv_usec = 500;

setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &time, sizeof(time));
```

- By default, socket timeouts are *disabled*
  - To *enable them*, timeout must be ≠0 (*tv_sec and/or tv_usec ≠ 0*)
  - To *disable them*, set timeout to *tv_sec=tv_usec=0*

Socket I/O TIMEOUT

- **SO_RCVTIMEO** modifies the behaviour of *all* readings from socket if the option was set with a timeout ≠ 0
  - Each *recv()/recvfrom()* call for the socket which has the option set will be *interrupted if it is not concluded before the timeout*
    - In such case the system call return value will be **-1**
    - **errno** will be set to *EAGAIN* and *EWOULDBLOCK* (to indicate the operation was interrupted before completion)
- **SO_SNDTIMEO** modifies the behaviour of *all* writings to the socket if the option was set with a timeout ≠ 0
- If *send()/sendto()* blocks for more than timeout, their return value will be **-1**
- **errno** will be set to *EAGAIN* and *EWOULDBLOCK* (to indicate the operation was interrupted before completion)

Socket kernel buffers

- Each socket has two dedicated buffers in the kernel:
  - **RECV buffer** where messages received by the socket and still not read by the application are kept
    - the queue of messages to be delivered to the application
  - **SEND buffer** where messages to be sent to the remote peer are kept
- socket receive/send default buffers size is *OS-dependent*

Linux	SO_RCVBUF	SO_SNDBUF
TCP	87380	16384
UDP	212992	212992

OS X	SO_RCVBUF	SO_SNDBUF
TCP	131072	131072
UDP	196724	9216

- Any application may change the default values to suit its needs:
  - **SO_RCVBUF** to modify the receive buffer size
  - **SO_SNDBUF** to modify the send buffer size

TCP and UDP output

- Maximum Transmission Unit (*MTU*) is the maximum packet size a link can transmit
  - Ethernet MTU =1500 bytes
- IPv4 and IPv6 define the *minimum reassembly buffer size*, i.e. the minimum datagram size all implementations must support
  - 576 bytes for IPv4, 1500 bytes for IPv6
- During 3WHS, TCP announces the Maximum Segment Size (*MSS*)
  - the maximum size of a TCP segment that the host will accept from the remote peer
  - On an Ethernet LAN, MSS values are usually:
    - 1500 bytes - 20 bytes IPv4 header - 20 bytes TCP header => 1460 bytes
    - 1500 bytes - 40 bytes IPv6 header - 20 bytes TCP header => 1440 bytes
- Recalling the above definition of *minimum reassembly buffer size*, the minimum value for MSS can be deduced:
  - 576 bytes IPv4 datagram - 20 bytes IPv4 header - 20 bytes TCP header => 536 bytes
  - 1500 bytes IPv6 datagram - 40 bytes IPv6 header - 20 bytes TCP header => 1440 bytes

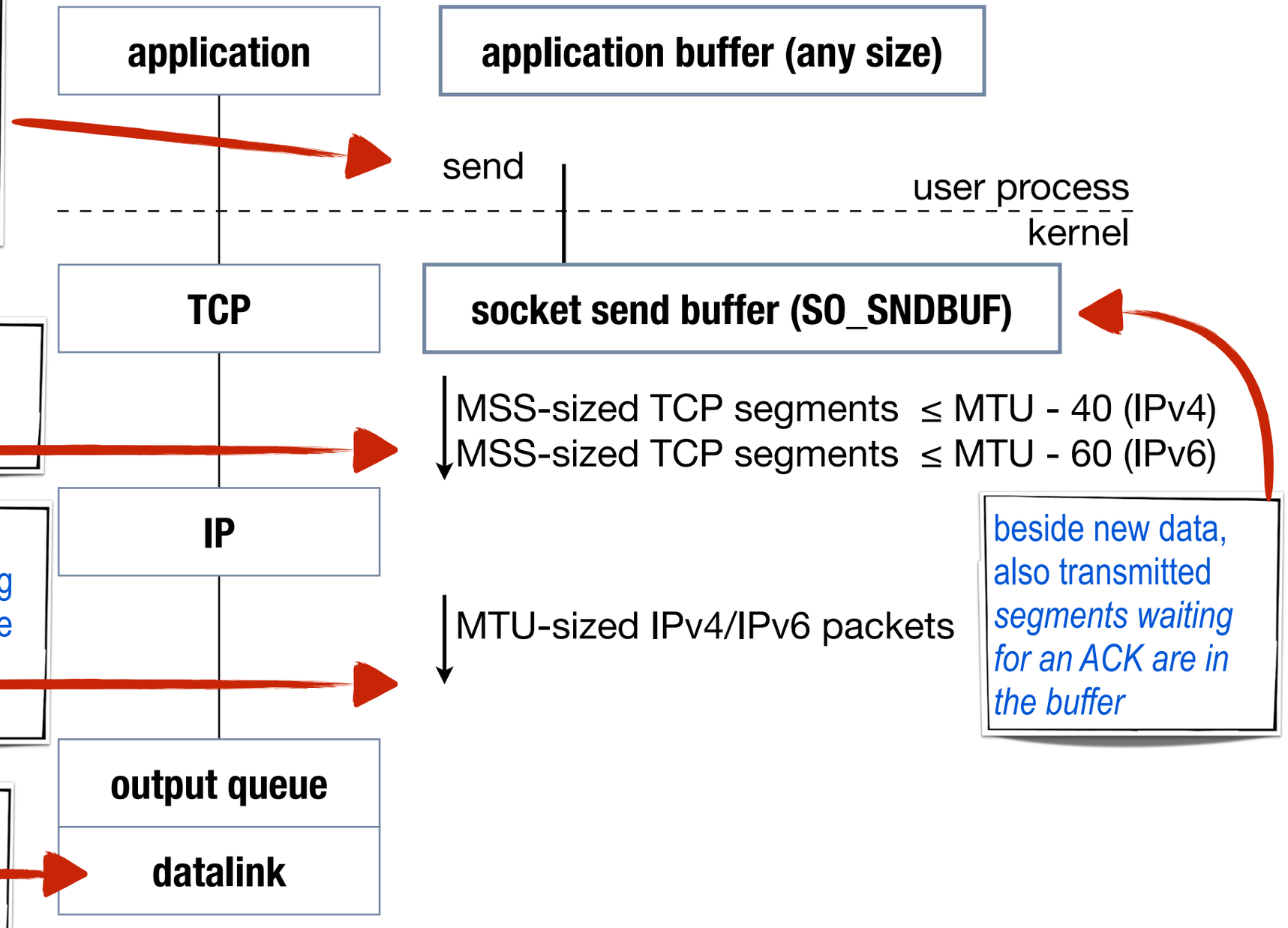
TCP output

The application issues a *send()* to the TCP socket: the kernel copies data from application buffer into the socket send buffer. If there is not enough space, the call blocks. The kernel returns from *send()* only after **all** bytes from the app buffer have been copied into the socket send buffer

TCP layer takes data from send buffer and packs them in segment of size **MSS** (agreed upon with 3WHS or fixes at *536 bytes*)

IP encapsulates the TCP segment in a **datagram**, reads in routing table the outgoing interface for the IP destination and hands the datagram to corresponding data link output queue

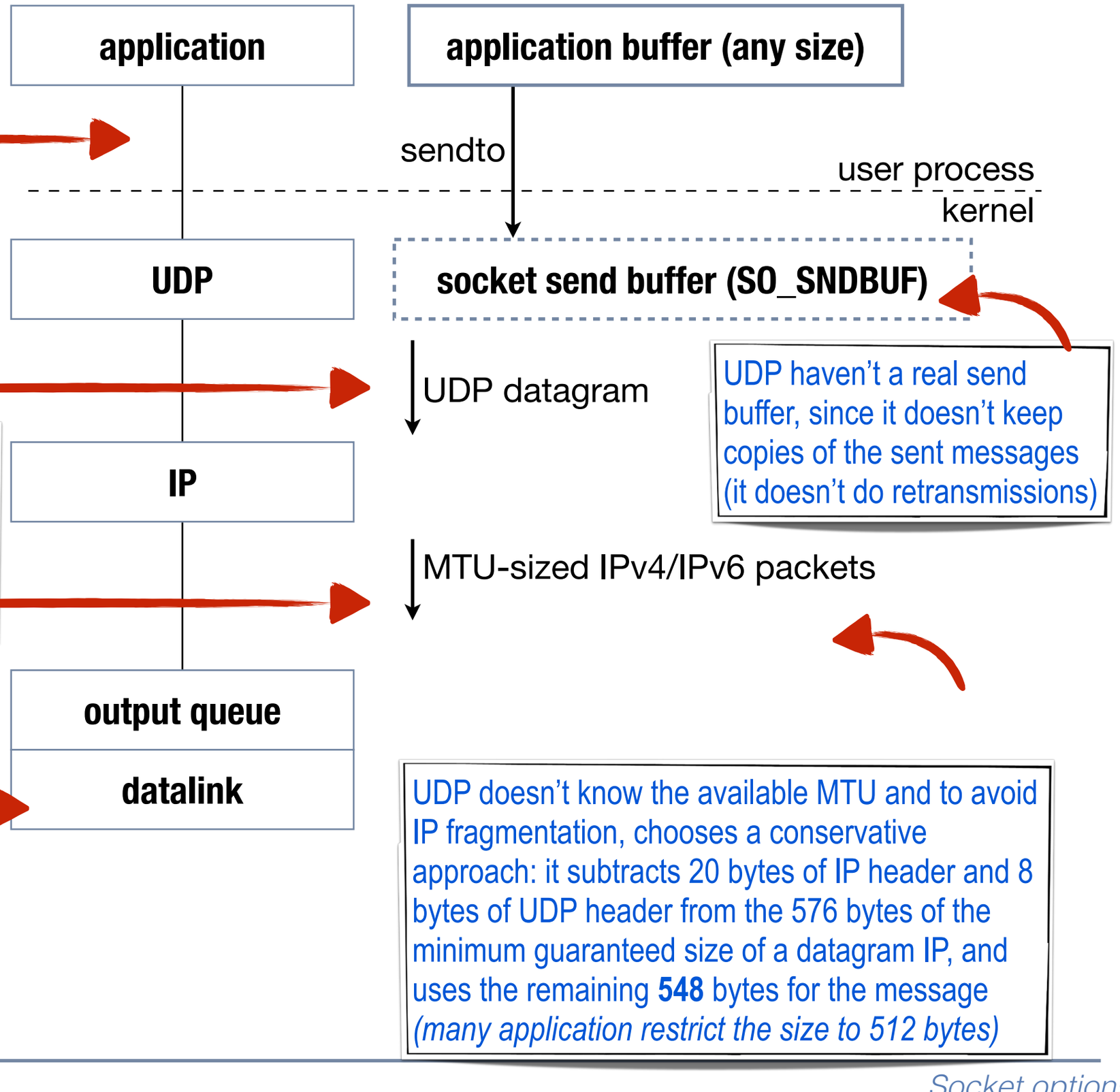
data link sends data in the output queue. If queue is full, packet is **discarded** and an error is returned to TCP. TCP will retransmit data, without notifying the application.



beside new data, also transmitted *segments waiting for an ACK are in the buffer*

UDP output

A UDP socket has a *send buffer size* (**upper limit** of the size of a message writable to the socket). If an application writes a message larger than the socket send buffer size, sendto() returns the **EMSGSIZE** error



UDP adds only an 8 bytes header to the datagram and hands it to IP

UDP haven't a real send buffer, since it doesn't keep copies of the sent messages (it doesn't do retransmissions)

IP encapsulates the **datagram**, reads in routing table the outgoing interface for the IP destination and hands the datagram to corresponding data link output queue (either entire or fragmented)

The data link takes the packets from the output queue and sends them. If the queue is full, the packet is **discarded** and the **ENOBUFS** error is returned to the application
(if the implementation supports it)

UDP doesn't know the available MTU and to avoid IP fragmentation, chooses a conservative approach: it subtracts 20 bytes of IP header and 8 bytes of UDP header from the 576 bytes of the minimum guaranteed size of a datagram IP, and uses the remaining **548** bytes for the message
(many application restrict the size to 512 bytes)

Receive Buffers

- The available space in the receive buffer of a TCP Socket limits the *window* size TCP communicates to the peer
  - *flow control* and *advertised window*: to avoid socket receive buffer overflow, each peer cannot send data beyond those a recipient is able to accept.
    - If a peer ignores the advertised window size and sends more data than those the receive buffer can hold, the receiving TCP will discard data in excess
- For a socket UDP: if an incoming datagram has a size greater than the available space in the receive buffer it will be discarded
- For a socket **TCP** the two option (SO_RCVBUF and SO_SNDBUF) must be set **before** beginning a new connection
  - info about *window scale option* are exchanged with the 3WHS
    - After 3WHS, changing those two option has no more effect
    - for a client, this means it must specify the receive buffer size *before connect()*
    - for a server, the receive buffer size must be specified for the listening socket *before listen()*
      - The *accepted socket* inherits the *listening socket* options

TCP Long Fat Pipe

- The sizes of the two peers' socket buffers affect the performances of a TCP connection
  - **connection throughput**
  - In the socket send buffer, beside fresh data queued by the application, there are also transmitted segments waiting to be ACKed
  - the receive buffer available size is the *advertised window.* Its value limits the peer's send rate
  - The *capacity of the TCP pipe* is defined as the *bandwidth-delay product* (BDP):

```
bandwidth(bit/s) x RTT(s) della connessione TCP
```

    - RTT is reported by a *ping*
    - the bandwidth is that of the slowest link between the two hosts
  - for slow connections (satellite link: RTT ~300ms) or for high capacity links, BDP is high => *TCP long fat pipe*
- If buffers' sizes are less than BDP, the TCP connection will not be able to exploit the *long fat pipe*, and throughputs will be lower than the available transfer rate.
- When BDP is high, buffer sizes must be incremented, so that the advertised TCP window will be automatically increased and better transfer rates are achieved

LOW WATER MARK

- Each socket has a *receive low-water mark* and a *send low-water mark*, used by *select()* to verify the sockets' *ready on read* and *ready on write* conditions
  - *receive low-water mark (**SO_RCVLOWAT**)* is the amount of *data* that must be present in the socket's receive buffer to have *select()* returning the socket as *readable*
    - the default value is **1 byte** for TCP and UDP
  - *send low-water mark (**SO_SNDLOWAT**)* is the amount of *free space* that must be present in the socket's send buffer to have *select()* returning the socket as *writable*
    - the default value is **2048 bytes** for TCP and UDP
      - transmitted segments waiting to be ACKed are kept in the send buffer
- *send low-water mark* for UDP, although defined, has different implications from the one for TCP:
  - UDP must not keep copies of sent datagrams, therefore the free space in the buffer doesn't depend on time (as it happens for TCP)
  - therefore, until the send buffer size is greater than the send low-water mark, the socket is always ready for writing

TCP socket option: TCP_NODELAY

- Among TCP level options (*level = IPPROTO_TCP*) only the **TCP_NODELAY** option is presently included in the SuSv4 standard, as defined in <netinet/tcp.h>
- It disables the *NAGLE algorithm*, whose purpose is to reduce the quantity of small packets sent on a TCP connection
  - *NAGLE*: If a connection sent data still not ACKed, the transmission of further small data packets is halted until the ACK of the previously sent data
    - After the ACK's reception, all data *buffered* until that moment will be sent in a *single TCP segment*
- In this way, the signalling overhead is optimized, by reducing the number of waited ACKs
  - This is desirable as long as the application doesn't suffer from the small delay caused by TCP buffering operations

IPV6: IPV6_V6ONLY

- For a socket in the *AF_INET6* domain the option **IPV6_V6ONLY** may be enabled so that the socket will accept **only** communications coming **from IPv6 addresses**
  - defaults to *OFF*
    - an IPv6 listening socket with this options set will accept only connections from IPv6 clients
    - refuses IPv4-mapped-IPv6 addresses

```
int v6flag=1;
//turn-on V6ONLY option
if (p->ai_family == AF_INET6 &&
   setsockopt(sockfd, IPPROTO_IPV6, IPV6_V6ONLY, &v6flag, sizeof(v6flag)) < 0)
   {
   perror("setsockopt error: ");
   close(sockfd);
   }//fi
```

Non-Blocking I/O

Catiuscia Melle and Franco Tommasi - SNP 2021-2022

Blocking vs Non-Blocking

- Sockets are **blocking**, by default and all system calls operating on them are blocking
  - If a socket call cannot be completed immediately, the process is put in sleep, waiting for the operation to conclude
    - example: a call to *recv()* on a socket returns only when data can be read from the socket receive buffer
    - the process is blocked on recv() until some data are available in the socket receive buffer
  - This behavior is sometimes undesired

- Socket calls can be classified in *4 groups*:
  - ***input operations*** (*read/recv/recvfrom*) blocking until data can be taken from the socket receive buffer
  - ***output operations*** (*write/send/sendto*) blocking until there is free space in the socket send buffer
  - ***accepting incoming connections*** (*accept*) blocking until a new connection is completed and a connection socket is made available to the process
  - ***initiating outgoing connections*** (*connect*) blocking until the 3WHS is completed

Blocking vs Non-Blocking

- Con un **nonblocking socket** invece si ottiene:
  - **input operations**: se non può essere completata, la call ritorna immediatamente con un errore EWOULDBLOCK
  - **output operations**: se non c'è spazio libero nel socket send buffer, la call ritorna immediatamente con un errore EWOULDBLOCK
  - **accepting incoming connections**: se invocata su un nonblocking socket, accept ritorna immediatamente con l'errore EWOULDBLOCK se la nuova connessione non è pronta
  - **initiating outgoing connections**: su un nonblocking socket connect ritorna immediatamente con queste condizioni di errore:
    - EINPROGRESS se il 3WHS è iniziato ma non ancora concluso
    - errore occorso, se la connect si è già conclusa con errore
    - Il procedimento di 3WHS intanto prosegue fino a completamento

Blocking vs Non-Blocking

- Sono possibili diverse alternative per evitare che un processo rimanga bloccato indefinitamente in una socket call per input o output operations:
  - utilizzare i *segnali* per interrompere operazioni bloccate (es. *alarm()* ed un'opportuno handler)
    - procedimento complesso - specie in ambito multi-thread
  - utilizzare le opzioni *SO_RCVTIMEO* e *SO_SNDTIMEO* per impostare un timer sulle operazioni di input e output di un socket
    - specifico per ogni socket
  - utilizzare *select()* per monitorare un gruppo di socket con un determinato valore di timeout
  - usare il *non-blocking I/O*
    - specifico per ogni socket

Non-Blocking I/O

- *fcntl* è la *file control function* che consente di impostare il file status flag *O_NONBLOCK* per un socket

```
#include <fcntl.h>

int fcntl(int fildes, int cmd, ...);
```

  - *fildes* è il descriptor su cui si desidera agire
  - *cmd* è il comando da utilizzare sul descriptor
    - **F_SETFL** consente di *impostare* un set di flag
    - **F_GETFL** consente di *leggere* il set di flag impostati

- Per un descriptor è definito un *set di flag*: per modificare il set si usano le operazioni logiche di *OR/AND* per aggiungere o rimuovere i nuovi flag
  - assegnare un singolo flag per volta cancellerebbe tutti gli altri dal set

Non-Blocking I/O: utilizzo del set di flag

- Il nonblocking I/O si imposta col flag ***O_NONBLOCK***:

```
int flags = fcntl(sockfd, F_GETFL, 0);

if (flags < 0)
{
  perror("F_GETFL error: ");
}
else
{
  //turn-on non-blocking I/O
  flags |= O_NONBLOCK;
  if ((fcntl(sockfd, F_SETFL, flags) < 0)
  {
    perror("F_SETFL error: ");
  }

  //do some stuff ....

  //turn-off non-blocking I/O
  flags &=~O_NONBLOCK;
  if ((fcntl(sockfd, F_SETFL, flags) < 0)
  {
    perror("F_SETFL error: ");
  }
```

OR logico per attivare il flag **O_NONBLOCK** nel set di flag prelevati dal descriptor

AND logico per disattivare il flag **O_NONBLOCK** (usando il valore negato **~O_NONBLOCK**)

Recuperare il set di flag impostati per quel descriptor

Impostare il nuovo set di flag

Reimpostare il set di flag precedente

Connect(): blocking vs nonblocking

- Potenziale scenario critico: un client TCP avvia la connessione ad un server TCP. In caso di errore, *connect()* ritorna il valore *-1* e la variabile *errno* varrà:

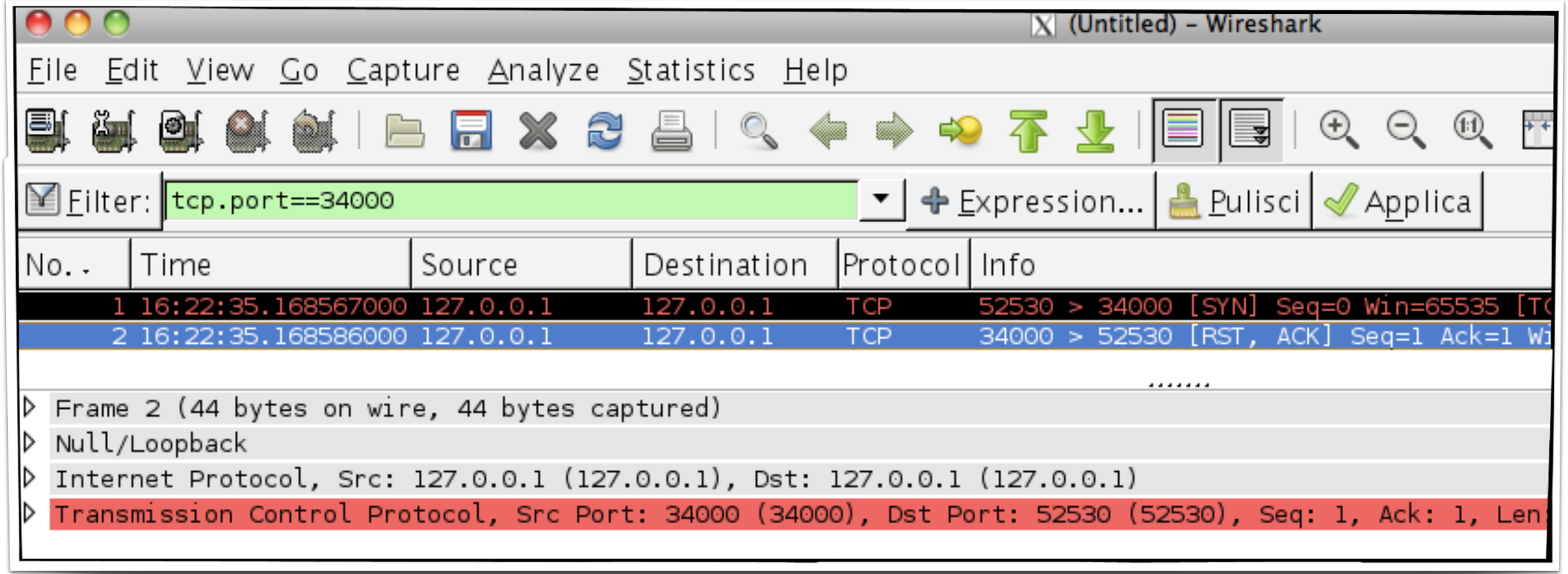
[EALREADY]:
A connection request is already in progress for the specified socket.
[ECONNREFUSED]:
The target address was not listening for connections or refused the connection request.
[EINPROGRESS]:
O_NONBLOCK is set for the file descriptor for the socket and the connection cannot be immediately established; the connection shall be established asynchronously.
[ENETUNREACH]:
No route to the network is present.
[ETIMEDOUT]:
The attempt to connect timed out before a connection was made.
[ECONNRESET]:
Remote host reset the connection request.
[EHOSTUNREACH]:
The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).
[ENETDOWN]:
The local network interface used to reach the destination is down.

Connect Blocking: Esempio 1

- Un client avvia una connessione verso un server *non attivo*:

```
$./testClient 127.0.0.1 hello
Connect()
Error code: 61 - Connection refused
Client failed to connect
```

- In risposta al ***SYN*** riceve un ***RST*** perché non c'è nessun servizio in ascolto sulla porta indicata.



Connect Blocking: Esempio 2

- Un client tenta di connettersi ad un server
  - il server è messo in sleep prima di aver effettuato *listen()* sul socket
  - entrambi i processi sono running su Mac OS X:
    - Mac OS X scarta il SYN ricevuto dal processo server in questo scenario, scatenando la ritrasmissione di SYN sul client
    - Wireshark ci mostra la sequenza di SYN che il client ha inviato al server in un intervallo di 75 secondi: solo a questo punto connect() ritorna con l'errore **ETIMEDOUT**

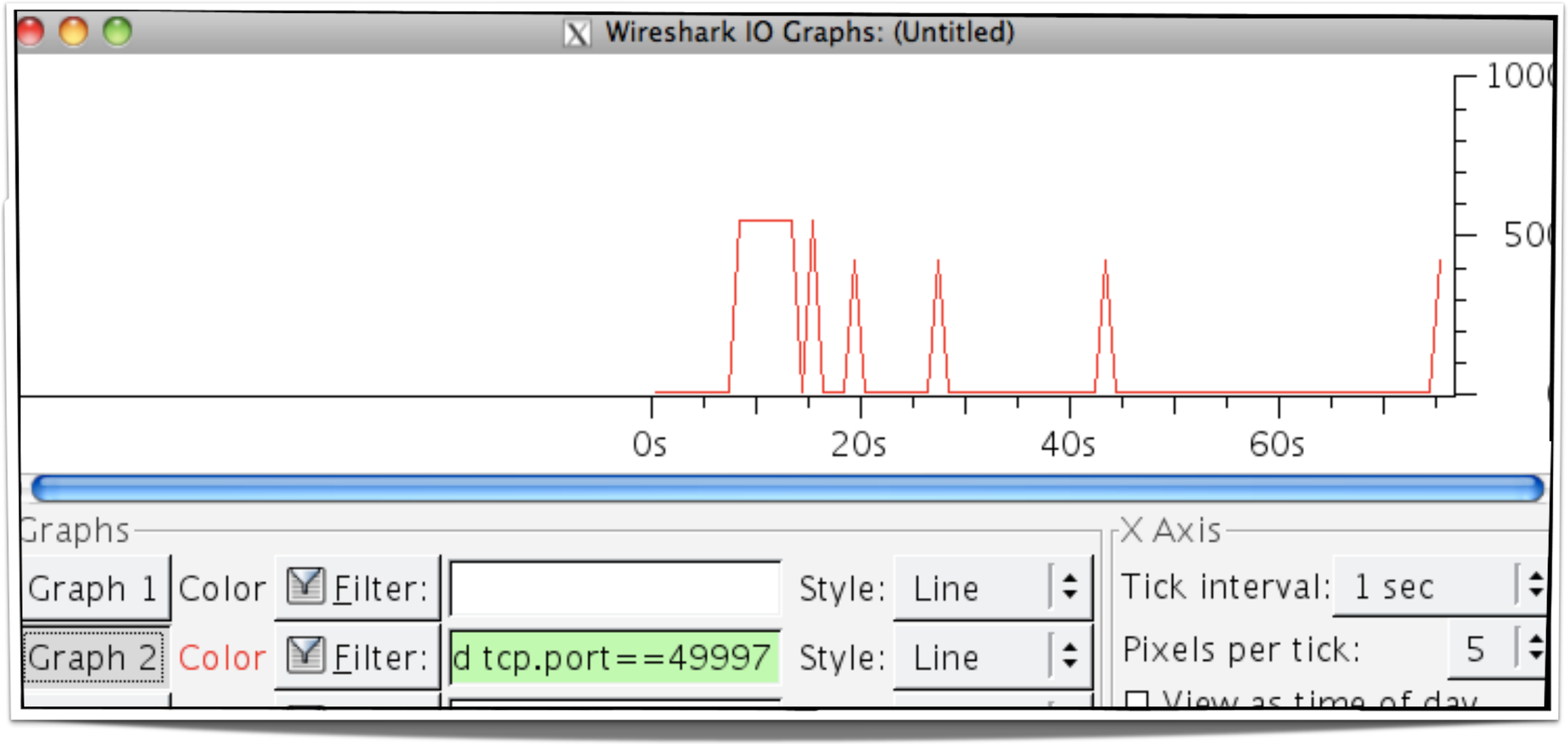
Filter: tcp.port==34000 | Expression... | Pulisci | Applica

No.	Time	Source	Destination	Protocol	Info
3	11:21:05.726490000	127.0.0.1	127.0.0.1	TCP	49997 > 34000 [SYN] Seq=0 Win=65535
4	11:21:06.672354000	127.0.0.1	127.0.0.1	TCP	49997 > 34000 [SYN] Seq=0 Win=65535
5	11:21:07.673233000	127.0.0.1	127.0.0.1	TCP	49997 > 34000 [SYN] Seq=0 Win=65535
6	11:21:08.674212000	127.0.0.1	127.0.0.1	TCP	49997 > 34000 [SYN] Seq=0 Win=65535
7	11:21:09.675108000	127.0.0.1	127.0.0.1	TCP	49997 > 34000 [SYN] Seq=0 Win=65535
8	11:21:10.676171000	127.0.0.1	127.0.0.1	TCP	49997 > 34000 [SYN] Seq=0 Win=65535
9	11:21:12.678137000	127.0.0.1	127.0.0.1		
10	11:21:16.681732000	127.0.0.1	127.0.0.1		
49	11:21:24.690138000	127.0.0.1	127.0.0.1		
127	11:21:40.701503000	127.0.0.1	127.0.0.1		
132	11:22:12.729321000	127.0.0.1	127.0.0.1		

```
$./testC 127.0.0.1 hello
Connect()
Error code: 60 - Operation timed out
Client failed to connect
```

Connect Blocking: Esempio 2

- Wireshark consente di estrarre il grafico del traffico generato dal client per la durata dell'operazione (*~75s*)
  - la frequenza di invio dei *SYN* nel client decresce col tempo
  - si inizia con l'invio di 1 SYN al secondo, poi ogni *2s*, *4s*, *8s*, *16s*, *32s*.



Connect Blocking: Esempio 2

- Ci sono diversi scenari in cui un tentativo di connessione può non andare a buon fine
- Fino allo scadere del timeout di sistema previsto per tale operazione, il processo rimane *bloccato* nell'esecuzione di *connect()*
- In alcuni casi, sarebbe desiderabile interrompere l'operazione **prima** dello scadere del timeout e proseguire con altre elaborazioni programmate
- L'unica alternativa per modificare il comportamento di *connect()* è ricorrere all'utilizzo del ***nonblocking I/O****
  - Impostando il nonblocking per il socket TCP **prima** della *connect()*, *connect()* ritorna immediatamente con l'errore ***EINPROGRESS*** mentre il protocollo TCP continua il 3WHS.
  - In un momento successivo, si potrà controllare lo stato della connessione utilizzando *select()*

Connect(): 3WHS

- Durante il 3WHS, l'instaurazione della connessione TCP tra due nodi implica che sul client *connect()* ritorni solo dopo che il *segmento ACK* è stato ricevuto in risposta al *segmento SYN* inviato
  - quindi *connect()* si blocca per un tempo pari ad **almeno 1 RTT** tra client e server
  - RTT può variare da pochi millisecondi in una LAN fino all'ordine dei secondi in una WAN
- Se la connessione non può completarsi con successo, solo allo scadere del timeout verrà riportato l'errore occorso:
  - il timeout varia da *75s* a diversi minuti - *implementation-dependent*
- Per eliminare questa attesa si può ricorrere all'utilizzo di *connect() non bloccante*
  - in questo modo sarà possibile per l'applicazione client eseguire altre operazioni attendendo l'esito del 3WHS
    - avviare più connessioni, impostare un tempo massimo d'attesa (application-dependent), …

Non Blocking Connect()

- Con una non-blocking connect() occorre considerare che:
  - *connect()* può anche ritornare immediatamente con successo (non solo con errore EINPROGRESS)
    - ad esempio, server e client sullo stesso host
  - Lo stato della connessione TCP si può valutare ricorrendo a *select()*, ma tenendo in conto un problema di portabilità tra le varie implementazioni.
  - Ad esempio per l'implementazione BSD e POSIX:
    - se la connessione si conclude con successo, il socket ritorna pronto in *scrittura*
    - se la connessione non può essere instaurata, il descriptor ritorna pronto sia in *lettura* che in *scrittura*

Server-side: Non Blocking Accept()

- *accept()* ritorna se c'è almeno una connessione completa
  - *select()* marka un listening socket *readable* se c'è una connessione completata da poter accettare
- Può accadere che una connessione venga instaurata e terminata prima che il processo server esegua accept().
  - un client avvia la connessione e subito dopo, grazie all'opzione SO_LINGER, invia un **RST** al server
- In questo caso, sebbene il listening socket sia *readable*, una chiamata ad accept sarà bloccata fino a completamento di un'altra connessione.
  - Questo scenario si può aggirare mettendo il listening socket in modalità nonblocking.

1-to-[many | all] Communications

Broadcast and Multicast

Catiuscia Melle and Franco Tommasi - SNP 2021-2022

Broadcasting and Multicasting

- Interaction models seen until now were based on 1-to-1 (**unicast**) communication
  - A pair of nodes (sender, receiver) exchanges messages following the rules of a predefined protocol
- For UDP sockets only, it is possible to set-up a **1-to-all** or a **1-to-many** form of communication
  - *(1-to-all) or* ***broadcasting***
  - *(1-to-many) or* ***multicasting***
- The advantage is obvious: it is possible to reach **many nodes** (selected on the base of a specific rule) at the same time by sending out a **single** datagram
  - The sender program is thus simplified and a better usage of the network is achieved

Broadcasting

- In a ***broadcast*** *communication*, a node sends a message to all nodes *of its network*, identified by the IP broadcast address.

- For IPv4, the address can be (N-bit two's complement representation for the host part):

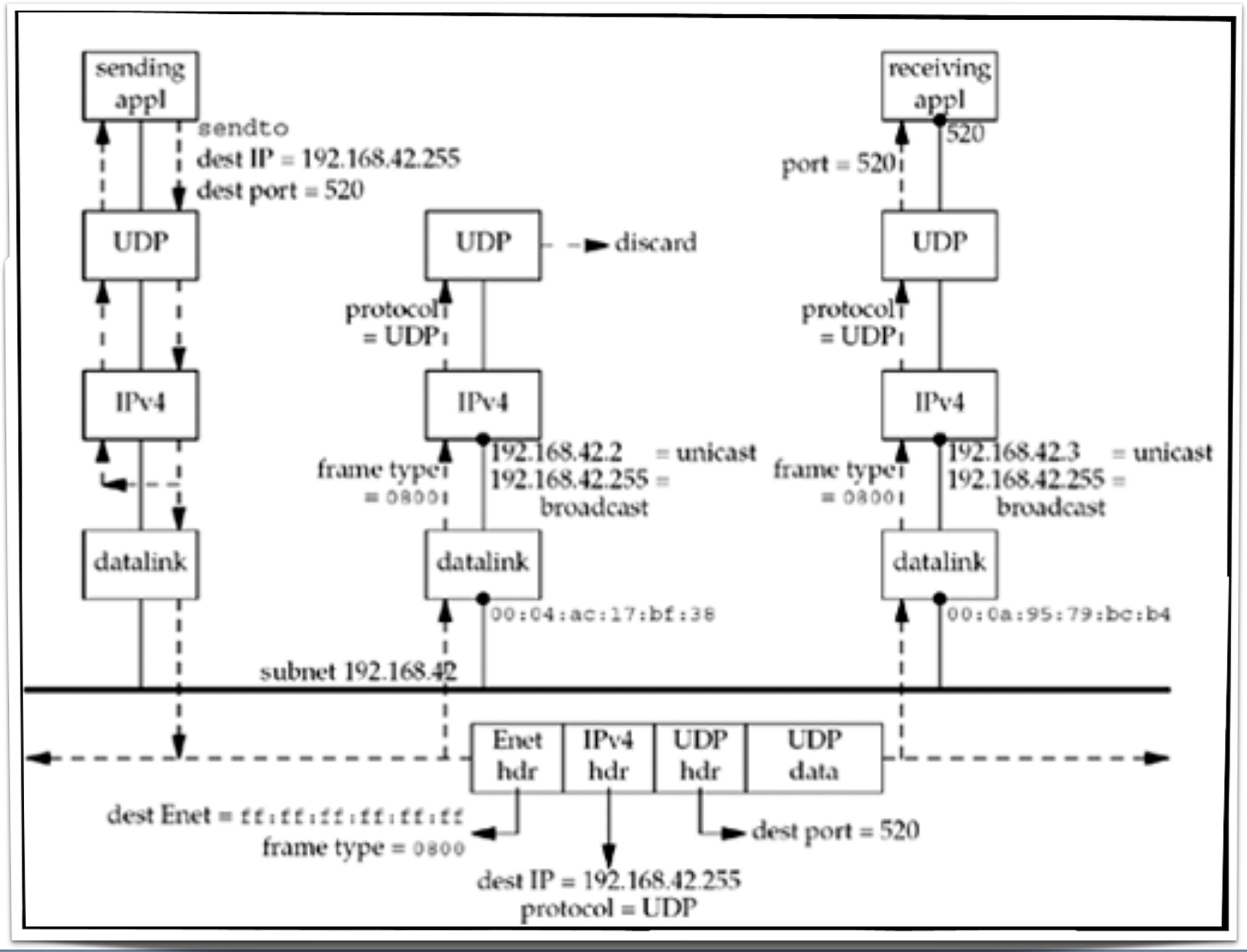
  - ***Subnet-directed broadcast address*****:** {subnetid, –1}
    Identifies al the interfaces in the specific subnet.
    For example, the broadcast address associated with the 192.168.42/24 subnet is *192.168.42.255*. A router should never forward a datagram sent to such address.

  - *Limited broadcast address*: {–1, –1, –1, -1} or 255.255.255.255
    Datagram to this address must not be forwarded by a router. Therefore they are addressed to al the host in the sender's subnet

    - *IPv6 does not support broadcast*: it supports a direct communication to all nodes of a network by multicast

Broadcasting

- A broadcast message is encapsulated in an E*thernet frame with a broadcast address and it is received by all* node in the LAN.



Broadcasting

- Since all interfaces (including those not interested in the message) must receive the broadcast frame, broadcasting implies a ***processing overhead for all hosts in a LAN***
  - the message will be discarded (in nodes not interested in the communication) *by* the *UDP layer* (after realizing no process is listening on that port)
  - Therefore a broadcast transmission is to be preferred only when it interests most nodes in the target LAN
- To allow a **sender** a broadcast transmission the generic option **SO_BROADCAST** must be enabled for the socket.
- No special setting in required for a **receiver**, but listening on the chosen port.

Multicasting

- Contrary to *broadcasting*,*multicasting* allows to specify *a set of destinations*, identified by a ***group address*** (*multicast*)
  - The IP multicast address does not identify univocally a network interface but rather a **set of nodes**
    - An *IPv4* multicast address belongs to the *class D*:
      - Class D addresses range from 224.0.0.0 to 239.255.255.255

```
1110xxxx . xxxxxxxx . xxxxxxxx . xxxxxxxx
```

    - An IPv6 multicast address is an address whose *first byte* is 0xFF
- Some IP multicast addresses are reserved. Their assignment is managed by IANA:
  - For IPv4
    - http://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml
  - For IPv6
    - http://www.iana.org/assignments/ipv6-multicast-addresses/ipv6-multicast-addresses.xhtml

Indirizzi IPv6

- IPv6 implements 3 types of addressing:
  - unicast (1:1),
  - multicast (1:M),
  - anycast (1:Any)
  - Among *unicast addresses* the following can be mentioned:
    - Loopback, **::1/128**: by this interface the node transmits to itself
    - Link-Local, **fe80::/10**: automatically configured on an interface when the IPv6 stack is started. They are identified by the first 10 bits, and are reserved for a link-local use (*not routable*)
    - Unique-Local, **fc00::/7** are identified by the first 7 bits. They correspond to IPv4 private addresses
    - Global-Unicast, **2000::/3** globally valid unicast addresses
- For multicast addresses, IPv6, beside the first byte 0xFF, uses another byte, split in two 4-bits fields: *4-bit flags* (00PT) e *4-bit scope*
  - If P=0, then:
    - T=0 => *IPv6 well-known multicast address* (0xFF0**x**::…)
    - T=1 => IPv6 transient multicast group (0xFF1**x**::…)
  - If P=1, then also T=1 (0xFF3**x**:…) indicates an IPv6 address based on a unicast prefix

Multicast addresses

- Among *reserved* IPv4 multicast addresses the following can be noted:
  - group *224.0.0.**1*** identifies **all the hosts** in a subnet
  - group *224.0.0.**2*** identifies **all the routers** in a subnet
  - and, more generally, the range from *224.0.0.0 to 224.0.0.255* is said ***link-local*** since it is reserved to protocols for management or discovery of *network topology*
    - this traffic is never forwarded by a router

- Among *reserved* IPv6 multicast addresses the following can be noted:
  - **ff01::1** the group of all nodes on an *interface-local* scope
  - **ff02::1** the group of all nodes on a *link-local* scope
  - **ff01::2**, **ff02::2**, **ff05::2** identifiy the group of **all routers**, respectively on an *interface-local*, *link-local* and *site-local* scope
    - *They are well-known IPv6 multicast addresses since the first byte is **0XFF**, followed by 4-bit flags equal to **0** (P=0 and T=0) plus the 4-bit scope (1,2,5)*

Multicasting

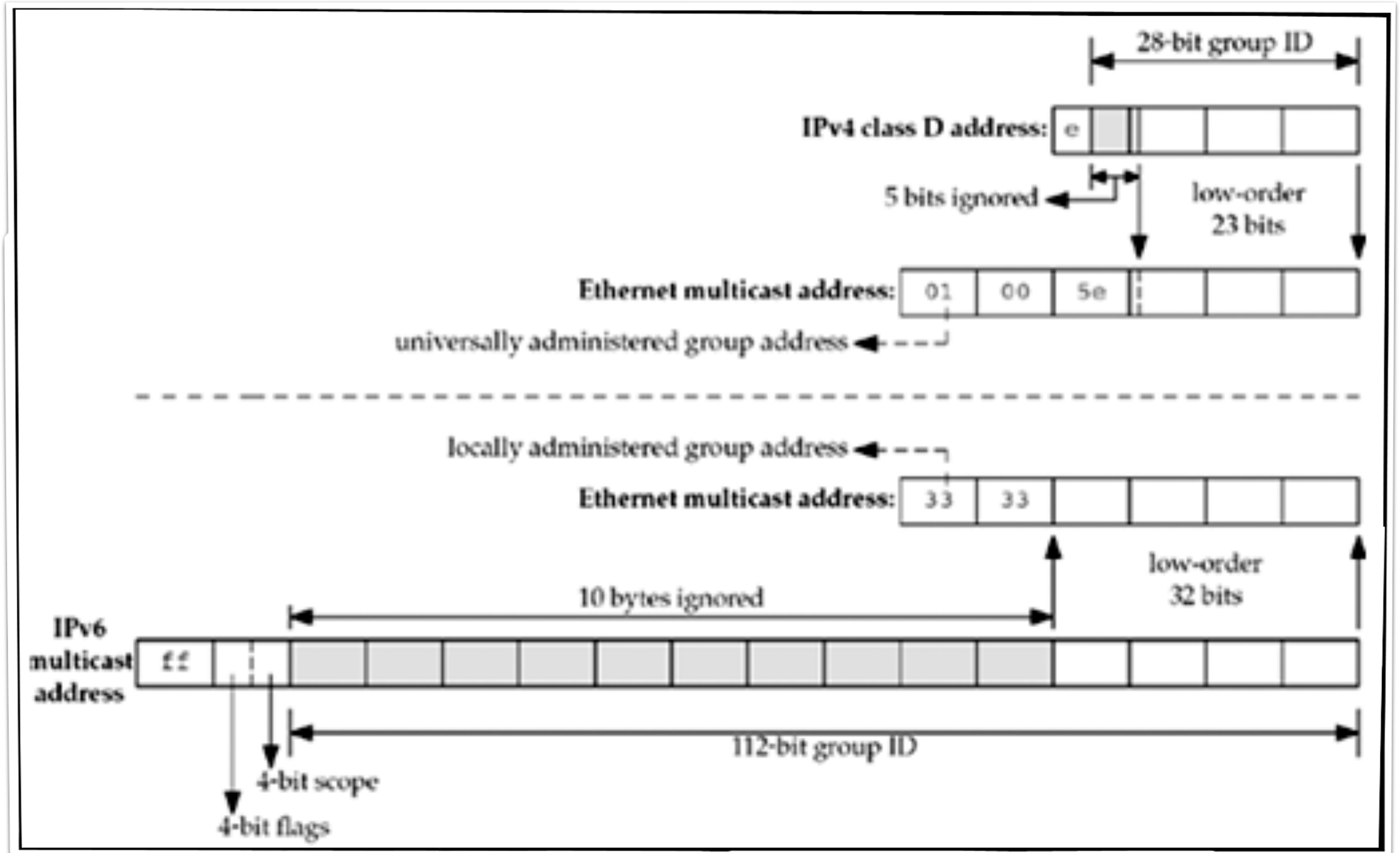
- Lo ***scope*** IPv6 determina quanto lontano un datagram multicast può andare
  - da non confondere con l'***hop limit*** che stabilisce quante volte può essere rilanciato da un router)
- *interface-local scope*: limitato al nodo
- *link-local scope*: limitato alla subnet di appartenenza del nodo
- *site-local scope*: limitato all'interno di un dominio (organization)
- In particolare, il gruppo di **tutti i nodi in link-local scope**, identificati dal gruppo ***FF02::1***, consente di contattare tutti i nodi di una subnet la cui interfaccia sia abilitata al traffico multicast.

Multicasting

- In IPv4 il concetto di *scope* è implementato in termini di valore del campo *TTL* dell'header IP:
  - *TTL 0*: interface-local scope,
  - *TTL 1*: link-local,
  - *TTL < 32*: site-local
  - *TTL < 128*: continent-local,
  - *TTL < 255*: global (unrestricted)

Multicasting

- L'indirizzo IP multicast viene convertito in indirizzo Ethernet multicast:



Multicasting

- Per IPv4:
  - dei 28 bit che identificano il gruppo multicast vengono presi i 23 di ordine più basso, concatenati ad altri 25 bit che identificano un indirizzo multicast Ethernet del tipo:
    - *01:00:5e:0*xxx.xxxx.xxxxxxxx.xxxxxxxx
- in questo mapping si tralasciano **5 bit**, ciò significa che ad ogni indirizzo multicast Ethernet corrispondono **32** diversi indirizzi IP multicast -> ***imperfect filtering***
- Per IPv6:
  - dei 128 bit vengono presi i 32 di ordine più basso e concatenati a 2 byte di valore 0x3333 per formare un indirizzo multicast Ethernet del tipo:
    - 33:33:yyyyyyyy.yyyyyyyy.yyyyyyyy.yyyyyyyy

Imperfect Filtering vs Perfect Filtering

- Poiché ad un indirizzo multicast Ethernet possono corrispondere più indirizzi IP multicast, l'elaborazione di un datagram multicast subisce questo iter:
  - a livello Ethernet, il confronto tra l'indirizzo di destinazione di un frame con l'indirizzo multicast è implementato con un *imperfect filtering*
  - il datagram estratto è consegnato al layer IP che esegue un ***perfect filtering***, perché confronta il destination address del datagram con l'indirizzo multicast

```
il pacchetto sarà scartato se i due IP non corrispondono
```

  - il layer UDP verifica che ci sia un processo in ascolto sulla porta di destinazione a cui consegnare il messaggio, altrimenti lo scarta

Inoltro Multicast

- L'invio di un messaggi multicast in una LAN subisce il trattamento illustrato.
- Più complesso è l'iter per l'instradamento in una WAN:
  - in questo scenario, il ruolo fondamentale è svolto dai router della rete
  - sono i router infatti a realizzare la comunicazione multicast, inoltrando una copia di un datagram multicast su tutte le interfacce differenti da quella da cui il pacchetto è arrivato
  - la segnalazione tra i router richiede il ricorso a complessi protocolli di routing, il cui scopo è creare un *albero di instradamento multicast*, che ha radice nel sender e raggiunge tutti i nodi appartenenti al gruppo
  - Si ottimizza così sia la fase di distribuzione del messaggio che l'utilizzo della bandwidth a disposizione

Inoltro Multicast

- Come si realizza un albero di instradamento multicast?
- Ogni nodo interessato ad una trasmissione multicast allerta il suo **first-hop router** dell'adesione al gruppo, così che il router possa accettare ed inoltrare nella subnet cui appartiene il nodo il traffico multicast ricevuto dagli altri router della rete.
  - Se non ci sono nodi appartenenti al gruppo, il router non inoltrerà copie di pacchetti su quella interfaccia
- La comunicazione tra i nodi di una subnet ed il first-hop router è realizzata mediante il protocollo **IGMP/IGMPv6**
  - un host (che intenda ricevere un traffico multicast) deve inviare un messaggio IGMP di *join/leave* al gruppo multicast al suo first-hop router

Multicast sender options

- Un sender multicast non ha delle operazioni particolari da effettuare: deve semplicemente conoscere l'identificativo del gruppo e la porta UDP a cui inviare i messaggi.
- Ha tuttavia 3 opzioni che consentono di modificare la trasmissione:

IP_MULTICAST_IF	***IPV6_MULTICAST_IF***
IP_MULTICAST_TTL	***IPV6_MULTICAST_HOPS***
IP_MULTICAST_LOOP	***IPV6_MULTICAST_LOOP***

Multicast sender options

- ***IP_MULTICAST_IF/IPV6_MULTICAST_IF***
  - consentono di specificare l'*outgoing interface* per il traffico multicast generato dal socket considerato.
  - Per IPv4, questa opzione richiede un valore di tipo *struct in_addr* che contenga l'indirizzo IP associato all'interfaccia scelta.
  - Per IPv6, l'opzione richiede un intero, che rappresenta l'*interface index*
  - Se come valori vengono passati *INADDR_ANY* e *0*, sarà il sistema a scegliere l'outgoing interface per ogni invio (leggendola dalla routing table del nodo)
  - L'*ingoing interface* è invece quella selezionata dal receiver come interfaccia da cui ricevere il traffico multicast.

Multicast sender options

- ***IP_MULTICAST_TTL/IPV6_ MULTICAST_HOPS***
  - Queste opzioni, rispettivamente per IPv4 ed IPv6, consentono di specificare il numero massimo di inoltri per un datagram multicast.
  - *valore di default = 1* per entrambe, limitando l'inoltro alla subnet locale

Multicast sender options

- ***IP_MULTICAST_LOOP/IPV6_ MULTICAST_LOOP***
  - Consentono di abilitare/disabilitare il local loopback dei datagram multicast
  - Se abilitato (*default on*), quando il sender invia un pacchetto, questo sarà looped-back
    - se il processo ha fatto il join al gruppo multicast, potrà ricevere una copia di ciò che invia
  - Attenzione, il tipo di valore da passare alle due opzioni è differente:

```
u_char option_loop = 0; per disabilitare il loopback in IPv4;
u_int option_loop = 0; per disabilitare il loopback in IPv6.
```

Ricezione Multicast

- A differenza di quanto accade col broadcasting, in cui è il *sender* ad abilitare la trasmissione broadcast agendo sul socket UDP, col multicasting è il *receiver* che deve effettuare le operazioni necessarie ad abilitare la ricezione sul socket.
  - Segnalazione IGMP, predisposizione del layer Ethernet, predisposizione del layer IP, per la ricezione multicast sono svolte dall'applicazione in modo *trasparente*, semplicemente invocando la funzione *setsockopt()* con le opzioni di livello *IPPROTO_IP/IPPROTO_IPV6*

- Le opzioni, rispettivamente IPv4 ed IPv6 sono:
  - *IP_ADD_MEMBERSHIP*, *IP_DROP_MEMBERSHIP*
  - *IPV6_JOIN_GROUP*, *IPV6_LEAVE_GROUP*

Invio e Ricezione Multicast

- Ecco come si presenta una sessione di traffico multicast

Filter: ip.addr==224.100.70.50 | Expression... | Pulisci | Applica

No.	Time	Source	Destination	Protocol	Info
1	19:58:44.060590000	192.168.1.100	224.100.70.50	UDP	Source port: 60749 Destination port: 51678
7	19:58:46.060938000	192.168.1.100	224.100.70.50	UDP	Source port: 60749 Destination port: 51678
15	19:58:48.061275000	192.168.1.100	224.100.70.50	UDP	Source port: 60749 Destination port: 51678
16	19:58:50.061625000	192.168.1.100	224.100.70.50	UDP	Source port: 60749 Destination port: 51678
23	19:58:52.061985000	192.168.1.100	224.100.70.50	UDP	Source port: 60749 Destination port: 51678
28	19:58:54.062359000	192.168.1.100	224.100.70.50	UDP	Source port: 60749 Destination port: 51678
29	19:58:56.062778000	192.168.1.100	224.100.70.50	UDP	Source port: 60749 Destination port: 51678
30	19:58:58.063061000	192.168.1.100	224.100.70.50	UDP	Source port: 60749 Destination port: 51678
33	19:59:00.063636000	192.168.1.100	224.100.70.50	UDP	Source port: 60749 Destination port: 51678

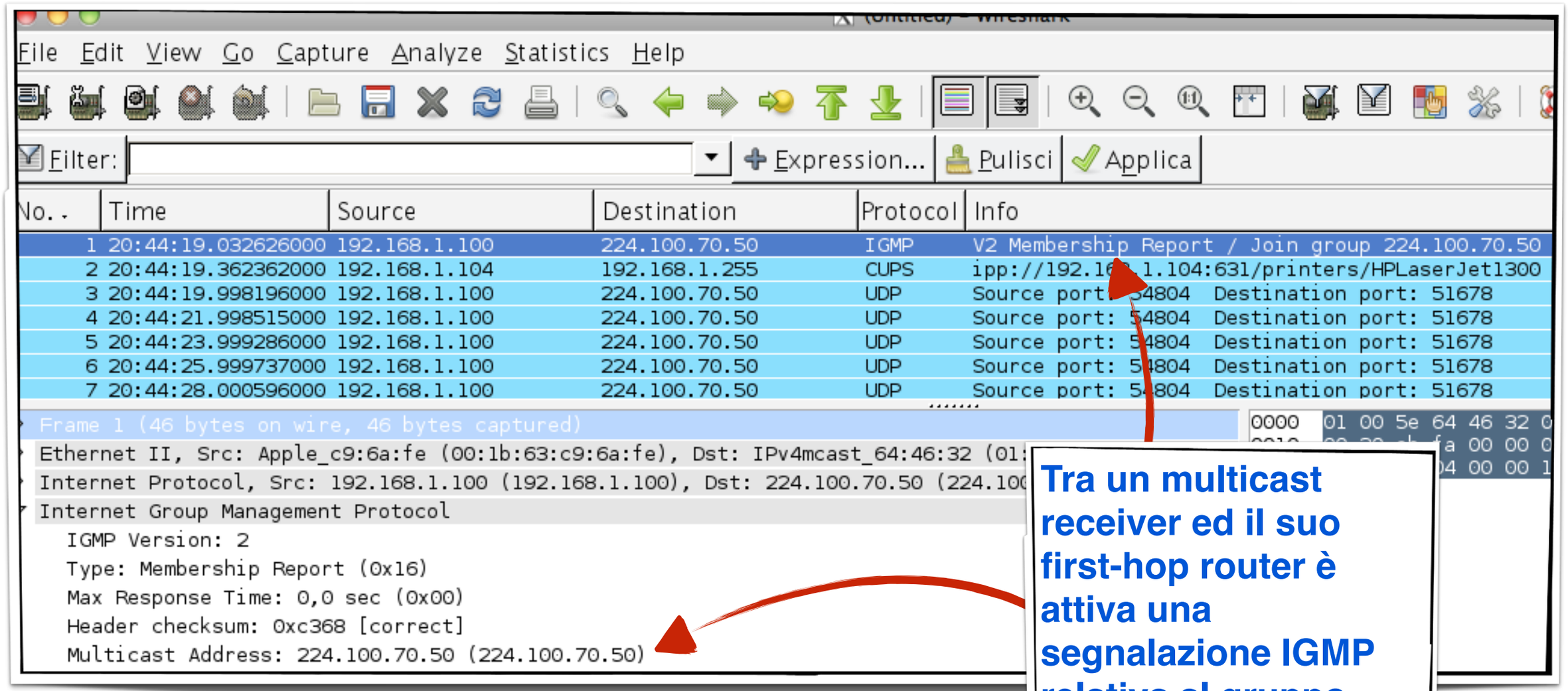
```
Frame 16 (60 bytes on wire, 60 bytes captured)
Ethernet II, Src: Apple_c9:6a:fe (00:1b:63:c9:6a:fe), Dst: IPv4mcast_64:46:32 (01:00:5e:64:46:32)
Internet Protocol, Src: 192.168.1.100 (192.168.1.100), Dst: 224.100.70.50 (224.100.70.50)
User Datagram Protocol, Src Port: 60749 (60749), Dst Port: 51678 (51678)
Data (18 bytes)
```

```
0000  01 00 5e 64 46
0010  00 2e fe a6 00
0020  46 32 ed 4d c9
0030  61 73 74 4d 65
```

La porta UDP è quella stabilita
L'indirizzo IPv4 è quello del gruppo multicast
Il MAC address è quello multicast ottenuto con la regola presentata nelle precedenti slide

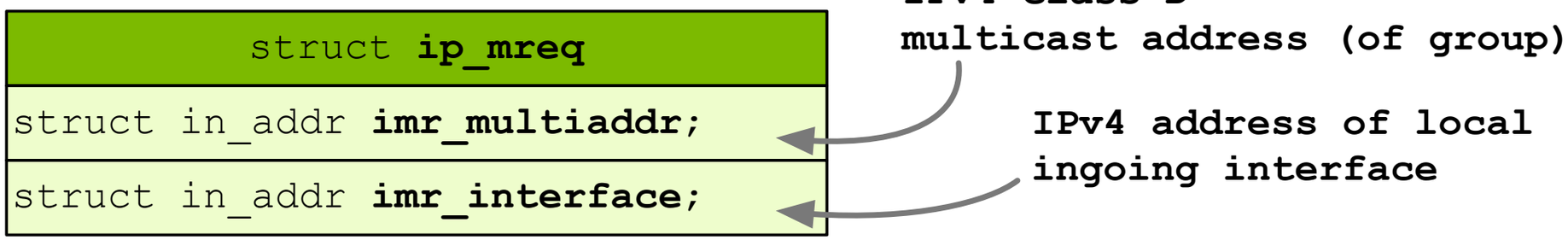
Invio e Ricezione Multicast

- Un *receiver multicast*, durante la fase di preparazione alla ricezione deve avviare una segnalazione *IGMP* col *first-hop router* per informarlo del suo *JOIN* al gruppo



Multicast receiver options: IPv4

- I*P_ADD_MEMBERSHIP* e *IP_DROP_MEMBERSHIP*
  - Per effettuare il *join* o il *leave* ad un gruppo multicast, un receiver deve conoscere l'indirizzo del gruppo ed eventualmente indicare l'ingoing interface da cui ricevere il flusso multicast
  - Si ricorre ad una struttura dati specifica per IPv4 che consente di raccogliere entrambe le informazioni:



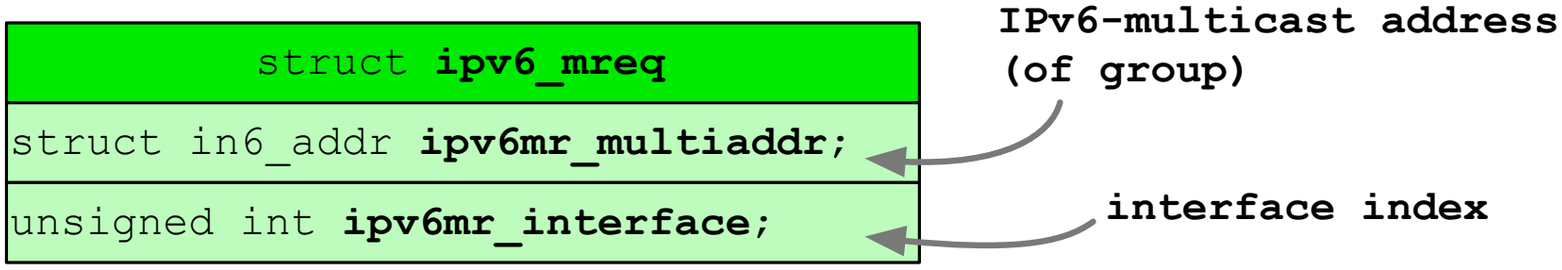
    - Specificando l'ingoing interface col wildcard address *INADDR_ANY*, è il sistema a sceglierla (consultando la routing table del nodo)
  - Al termine della procedura di join, si dice che *il nodo appartiene al gruppo multicast sulla interfaccia specificata*

Multicast receiver options: IPv4

- Un socket può eseguire più join, a patto che ognuna indichi un diverso gruppo multicast o una differente interfaccia (a parità di multicast IP).
  - Il numero massimo di join consentite per un socket è specificato dalla costante ***IP_MAX_MEMBERSHIP***, il cui valore è implementation-dependent.
- L'abbandono di un gruppo multicast segue lo stesso procedimento, ma con l'opzione ***IP_DROP_MEMBERSHIP***
  - Il leave del gruppo è automatico alla chiusura del socket o alla terminazione del processo di ricezione.
  - Se un nodo ha fatto il join ad un gruppo multicast con più socket, esso rimarrà membro del gruppo fino alla chiusura dell'ultimo socket.

Multicast receiver options: IPv6

- *IPV6_JOIN_GROUP* e *IPV6_LEAVE_GROUP*
  - A differenza di IPv4, questa opzione necessita della struttura:



  - in cui, oltre a specificare un **indirizzo IPv6 multicast**, il principale cambiamento è l'utilizzo di un **intero** per specificare la local ingoing interface invece dell'indirizzo IP
  - Poiché gli indirizzi IPv6 multicast che differiscono solo per il valore di ***scope*** rappresentano **gruppi differenti**, se un receiver desidera ricevere **tutto** il traffico multicast *indipendentemente dallo scope*, dovrà eseguire sul socket il join a tutti i gruppi:
    - *ff01::x* (**node scope**), *ff02::x* (**link scope**), *ff05::x* (**site scope**), *ff08::x* (**organization scope**), *ff0e::x* (**global scope**).

Interface index

- Per recuperare l'interface index da utilizzare con le opzioni IPV6_JOIN_GROUP/IPV6_LEAVE_GROUP:

```
#include <net/if.h>

unsigned if_nametoindex(const char *ifname);
```

- a partire dal nome dell'interfaccia (*en0, eth1, lo0,..*) ritorna l'**unsigned** che ne rappresenta l'indice

- La funzione inversa:

```
#include <net/if.h>

char *if_indextoname(unsigned ifindex, char *ifname);
```

- in cui *ifname*, che punta ad un buffer di dimensione ***IF_NAMESIZE***, conterrà al ritorno il nome dell'interfaccia il cui indice è *ifindex* (**NULL** in caso di errore)

Receiver operations

- Affinché un processo possa ricevere un traffico multicast, deve quindi
  - deve eseguire il **bind** del socket alla porta di destinazione scelta per il gruppo
  - effettuare il **join** al gruppo multicast
- Il *join* è necessario affinché l'IP layer ed il datalink layer si predispongano alla ricezione del traffico multicast (avviando la segnalazione IGMP al first-hop router)
- Il *bind* serve per indicare al layer UDP che si desiderano ricevere quei datagram
  - eseguendo il bind completo (IP+porta) si è certi che solo i datagram richiesti saranno consegnati al socket (e non anche pacchetti diretti alla stessa porta ma per un altro indirizzo IP)