NETWORK SOCKETS

INTRODUCTION TO NETWORK SOCKET PROGRAMMING AND CONCEPTS

Peter R. Egli
peteregli.net

Contents

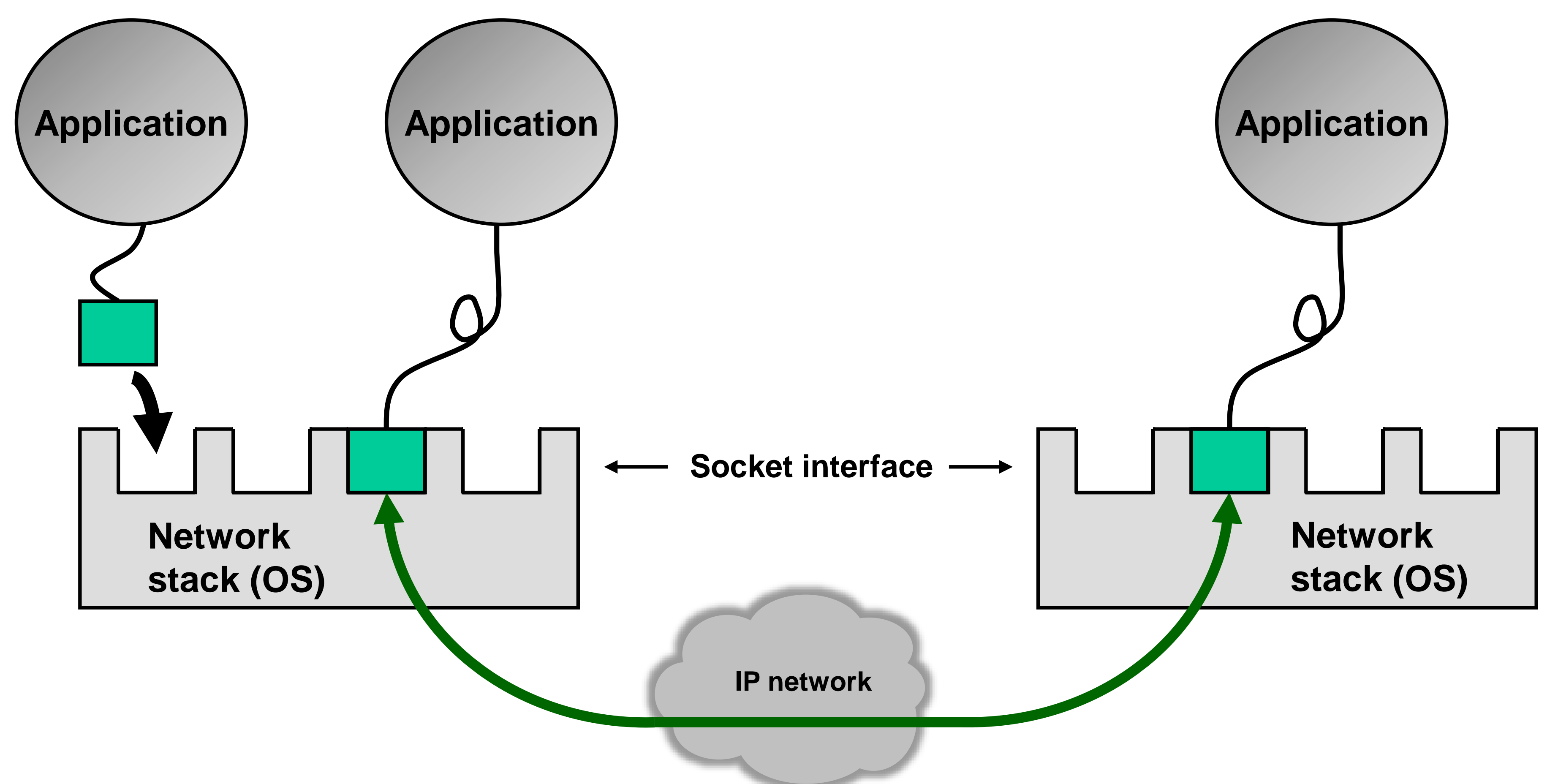
1. What is a socket?
2. Socket = Interface to transport API
3. Routing in Network Layers
4. TCP socket „spawning“
5. Socket interface functions
6. Socket calls versus TCP segments
7. Socket calls versus UDP datagrams
8. Socket handle
9. Parameter marshalling / RPC transparency
10. Low level socket programming
11. UDP multicast sockets
12. TCP server socket: C/C++ versus Java example
13. Client socket: C/C++ versus Java example
14. IPv6 sockets

© Peter R. Egli 2017

1. What is a socket?

A socket is an interface for an application to connect to a host‘s network stack (part of the OS).

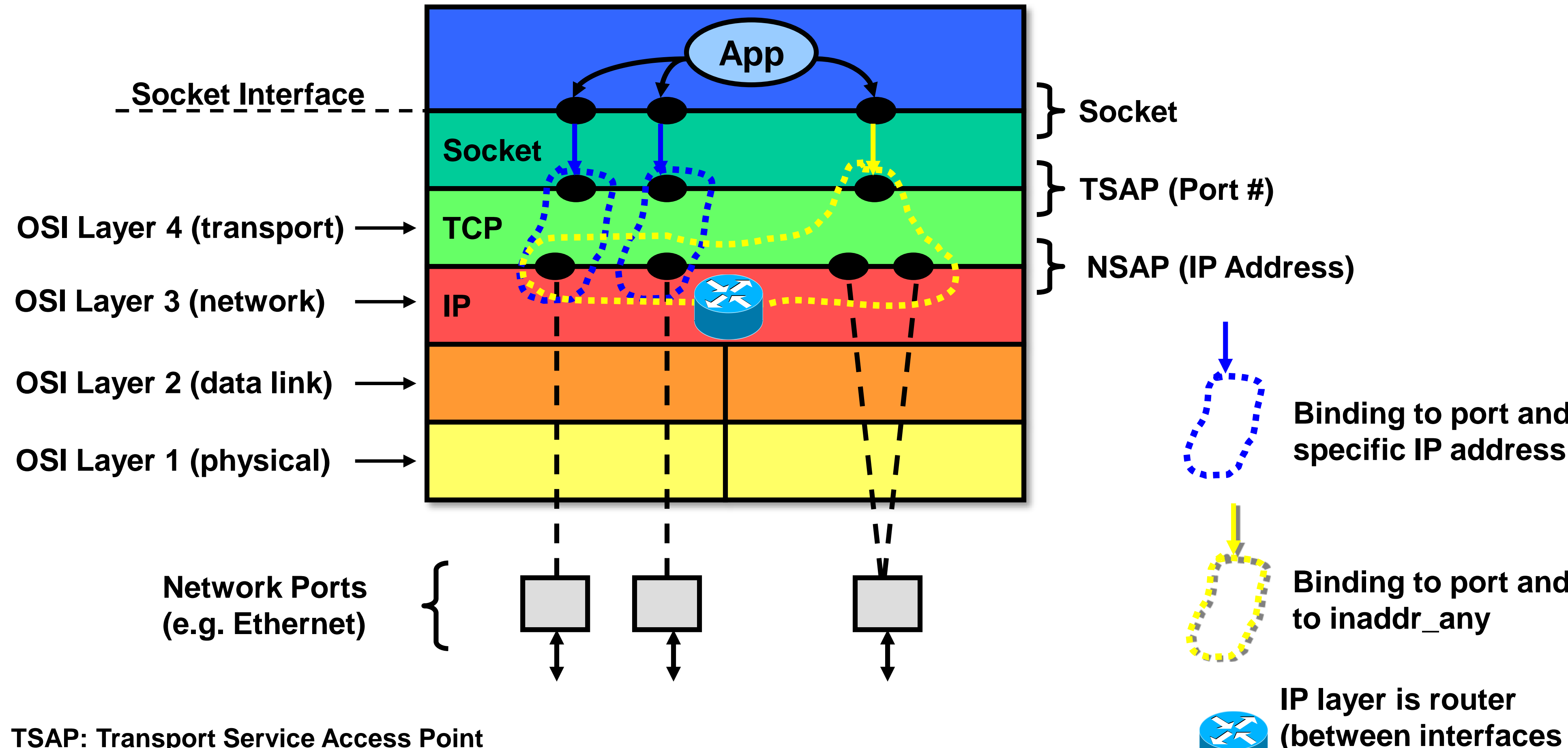
After connecting, an application is able to bidirectionally exchange data with other processes on the same or another host.



© Peter R. Egli 2017

2. Socket = Interface to transport API (host's transport protocols)

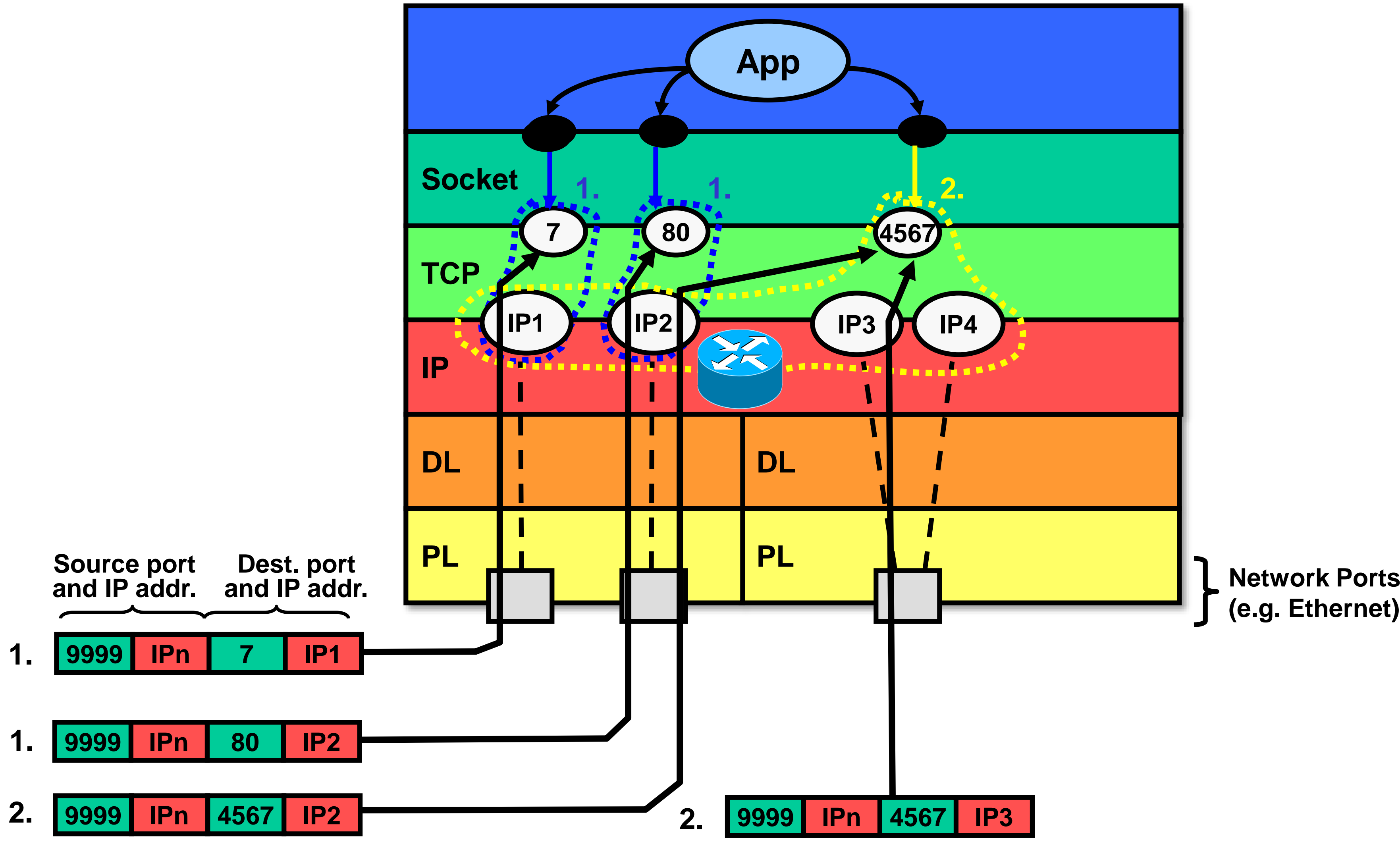
A socket has a binding to an NSAP (with an IP address) and a TSAP (with a TCP/UDP/SCTP port number). The NSAP may have a specific IP address or may represent all IP addresses of the host (unspecified IP address = wildcard address = 0.0.0.0 = inaddr_any).



TSAP: Transport Service Access Point
NSAP: Network Service Access Point

3. Routing in Network Layers (1/4)

The routing of packets from and to a socket depends on the bind IP address.



© Peter R. Egli 2017

3. Routing in Network Layers (2/4)

1. Specific IP address binding:

UDP socket:

If a UDP socket is bound to a specific IP address, only IP packets with this destination IP address are routed to and received by this socket.

TCP socket:

In case of a listening TCP socket, only connection requests (inbound connection) addressed to the bind IP are accepted by the socket.

2. inaddr_any binding:

If a socket is NOT bound to a specific IP address (INADDR_ANY = 0.0.0.0, wildcard IP address), the socket is bound to all existing interfaces.

UDP socket:

A UDP socket receives any packet that contains the bind port number as target port.

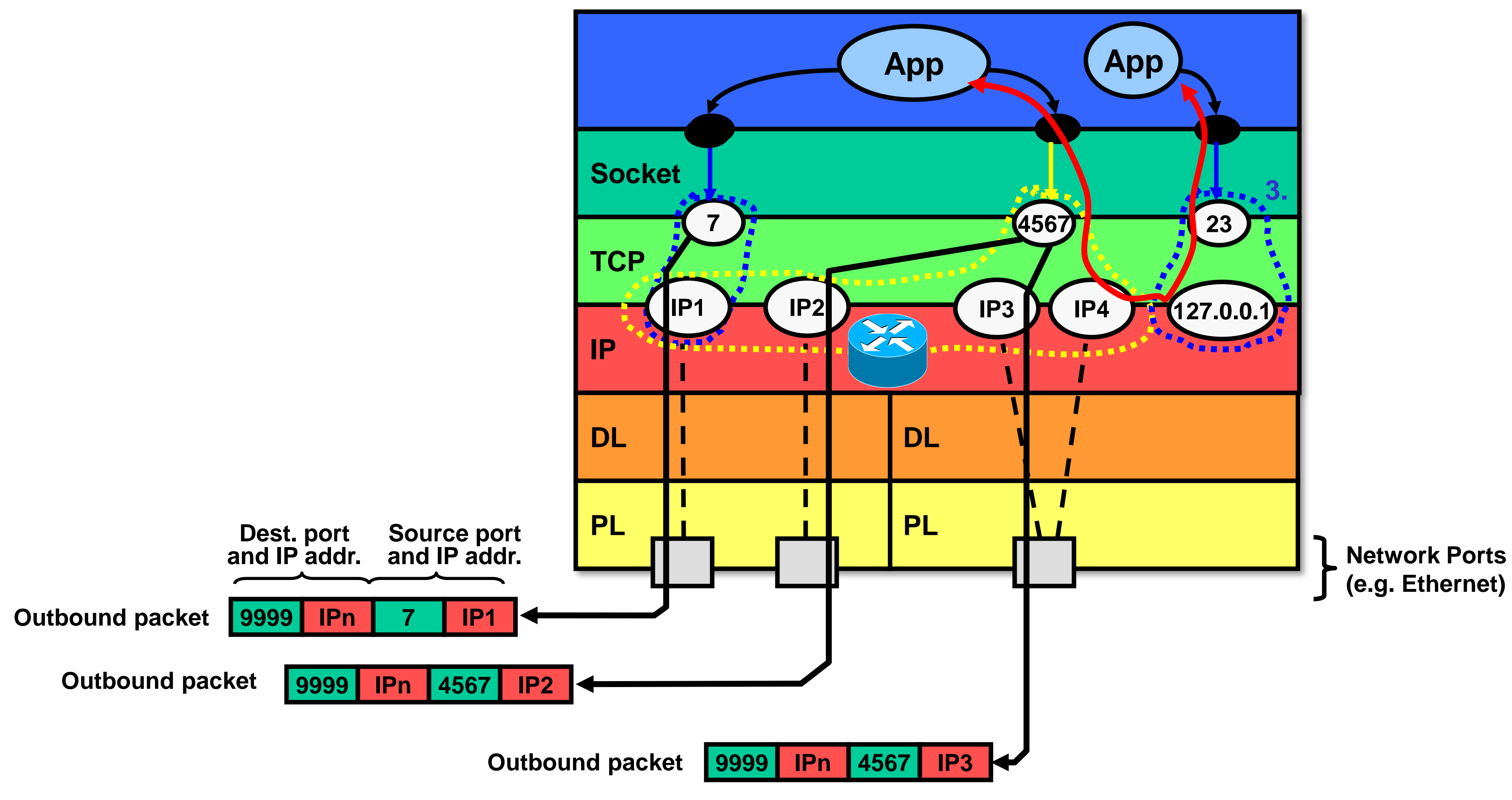
TCP socket:

A listening TCP-socket bound to 0.0.0.0 is able to accept connections on all interfaces provided that the destination port of the incoming connection request equals the bind port number.
Once the incoming connection is accepted, the created TCP-socket is bound to the destination IP address of the incoming connection request.

© Peter R. Egli 2017

3. Routing in Network Layers (3/4)

Localhost binding and routing of outbound packets:



© Peter R. Egli 2017

3. Routing in Network Layers (4/4)

3. localhost binding:

If a socket is bound to “localhost”=127.0.0.1, then this socket receives only from applications but not from the network.
Besides the local loopback interface (127.0.0.1 for IPv4, ::1 for IPv6), applications on the same machine can also use an interface IP address for communication.

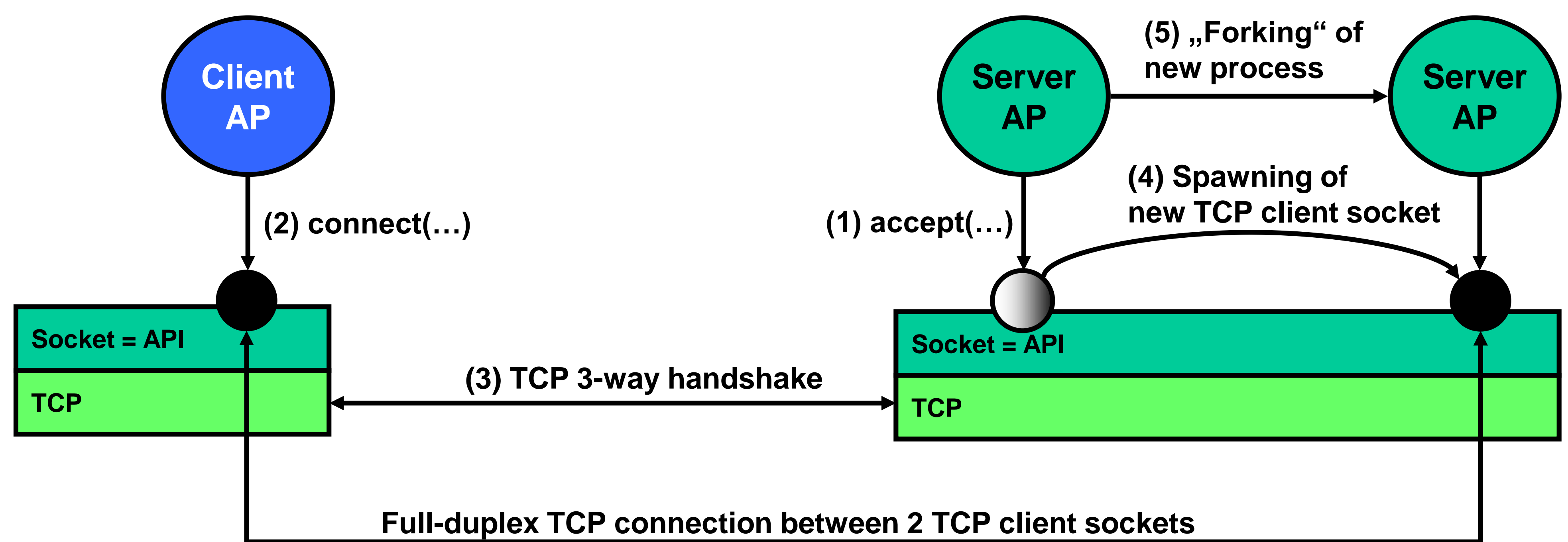
4. Outbound IP address:

The source address of outbound packets is either the bound IP address or the address of the interface over which the packet is sent (if the socket is bound to INADDR_ANY).
N.B.: An outbound packet may also be sent over an interface other than the socket is bound to, i.e. the routing is based on the IP layer’s routing table.

© Peter R. Egli 2017

4. TCP socket „spawning“

In TCP there exist 2 different socket types: server socket and client socket.
The server socket is used to accept incoming connections. When TCP receives an incoming connection request on a server socket (SYN) it spawns a new (client) socket on which the server process can send and receive data (after passing the new socket to a newly „forked“ server process).



5. Socket interface functions (1/2)

➔ **TCP Socket Interface Functions:**

Depending on the platform (Java, C, Python ...) client and server sockets may be implemented differently. In C (BSD sockets, Winsock) there is only 1 socket type while in Java client and server sockets are represented by different classes.

Client:

socket()	Create client socket
connect()	Create a connection
send()	Send data
receive()	Blocking receive data
close()	Close client socket

Server:

serversocket()	Create server socket
bind()	Bind server socket to socket address (IP+port)
listen()	Create queues for requests
accept()	Block on incoming requests
close()	Close server socket

© Peter R. Egli 2017

5. Socket interface functions (2/2)

➔ **UDP Socket Interface Functions:**

Client and server have the same socket functions.
There are no functions for connection setup / shutdown since UDP is connectionless.
With one UDP socket it is possible to send to different destination hosts (sendTo() function).

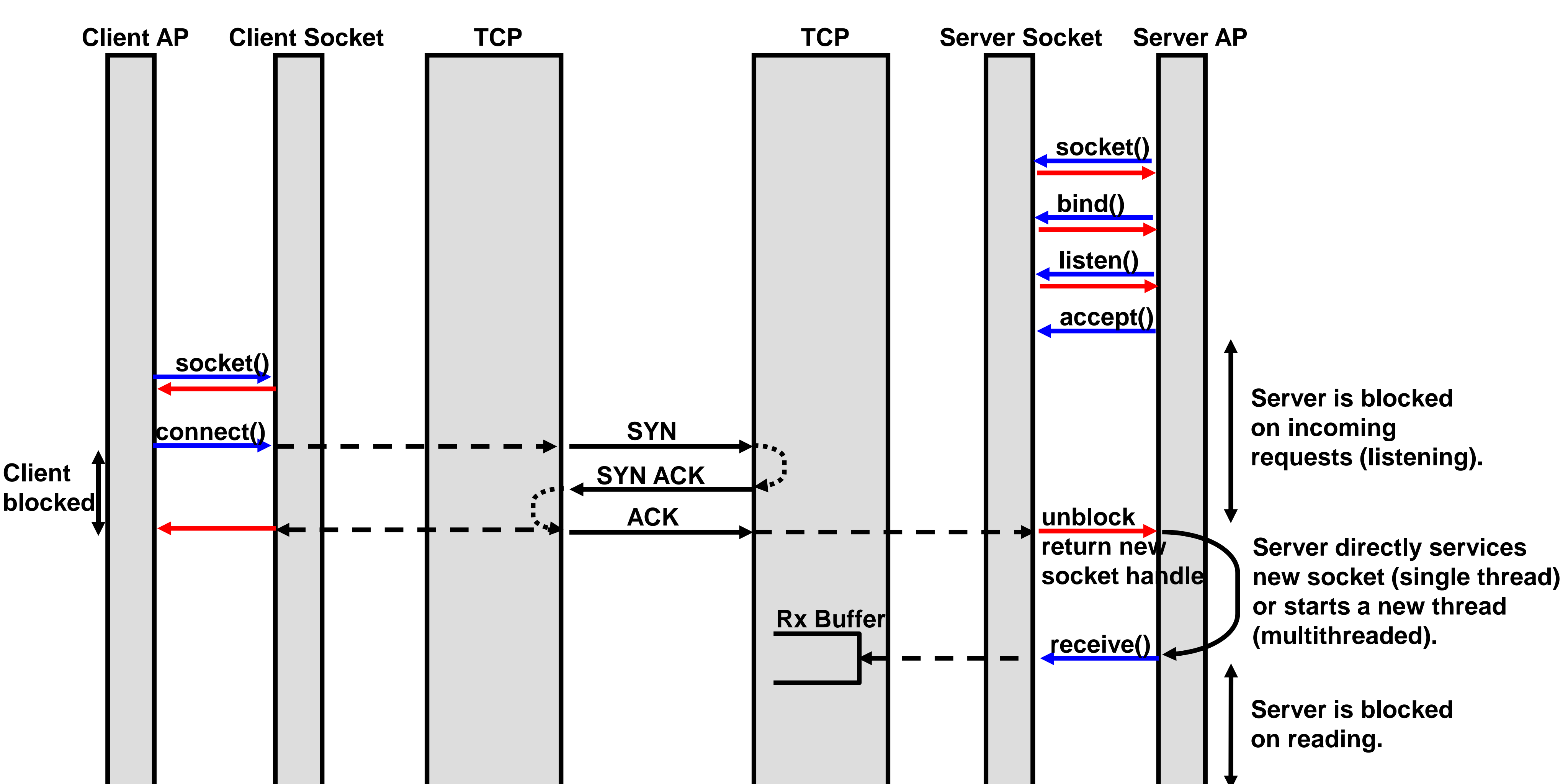
Client & Server:

socket()	create client / server socket
bind()	bind client / server to socket address (IP+port)
send()	send data (client and server)
receive()	receive data (client and server)
close()	close client / server socket

© Peter R. Egli 2017

6. Socket calls versus TCP segments (1/3)

➔ Connection establishment:



© Peter R. Egli 2017

6. Socket calls versus TCP segments (2/3)

➔ Socket send / receive (symmetric for client and server):

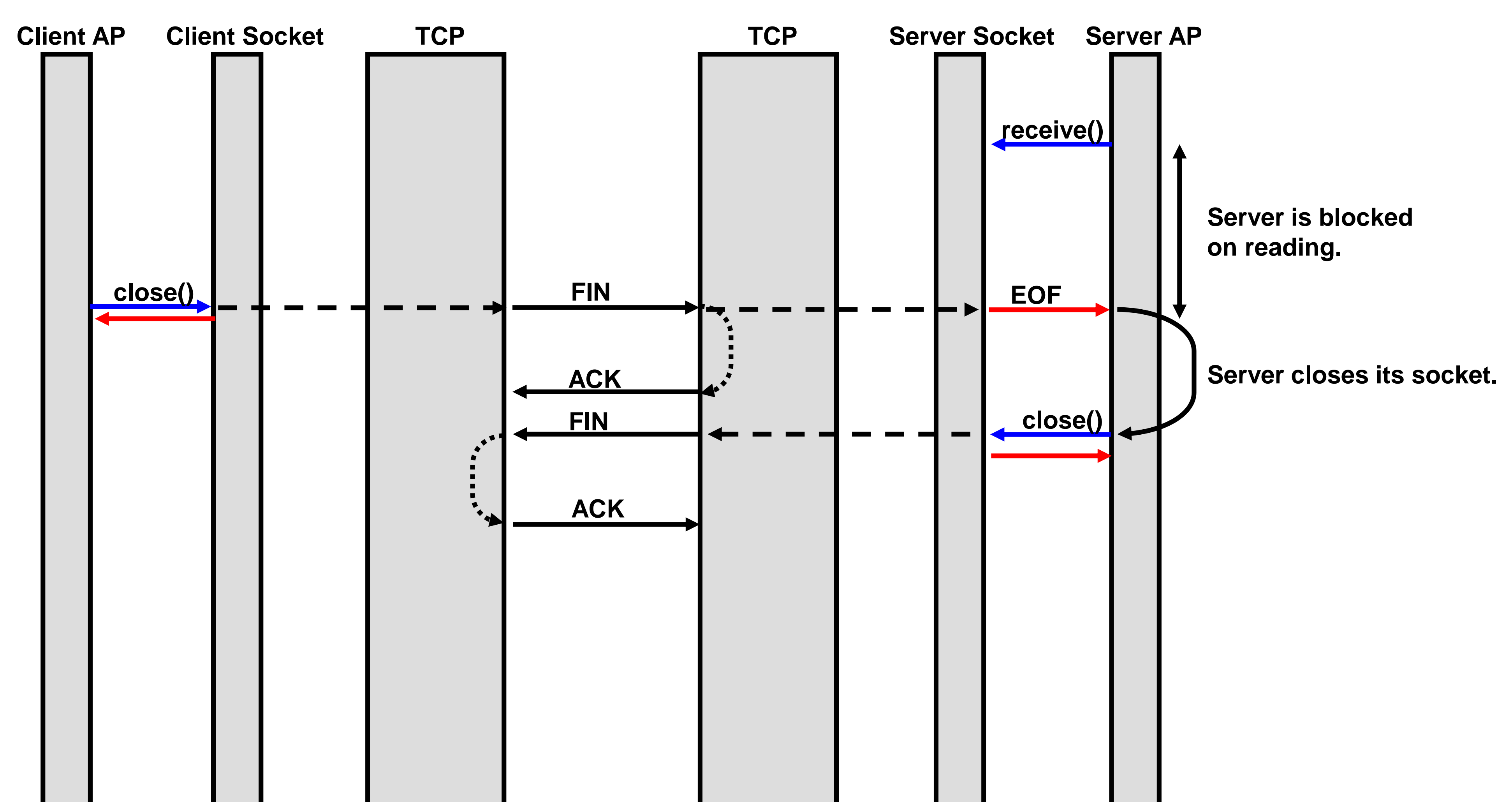
Function call and function return

Client AP
Client Socket
TCP
TCP
Server Socket
Server AP
Rx Buffer
receive()
Tx Buffer
send()
Rx Buffer
Data Segment
ACK
Server is blocked on reading.
Rx Buffer
unblock receive
Server handles the request.

© Peter R. Egli 2017

6. Socket calls versus TCP segments (3/3)

➔ Socket close:



© Peter R. Egli 2017

7. Socket calls versus UDP datagrams

Function call and function return

Client AP
Client Socket
UDP
UDP
Server Socket
Server AP

socket()
bind()
socket()
bind()
receive()
send()
Datagram
Rx Buffer
receive()
Rx Buffer
Server is blocked on reading.
Rx Buffer
Datagram
send()
Server handles the request.
Rx Buffer
unblock read
close()
close()

© Peter R. Egli 2017

8. Socket handle

In Unix a socket is like a file descriptor.

➔ Same handling as file (open, close, EOF).

➔ Input stream / output stream to read / write to / from socket (like file).

File:

```
fhdl = fopen(filename,"rw");
while not (EOF) {
  s = gets(fhdl);
}
puts(fhdl,"hello");
fclose(fhdl);
```

Socket:

Socket
TCP

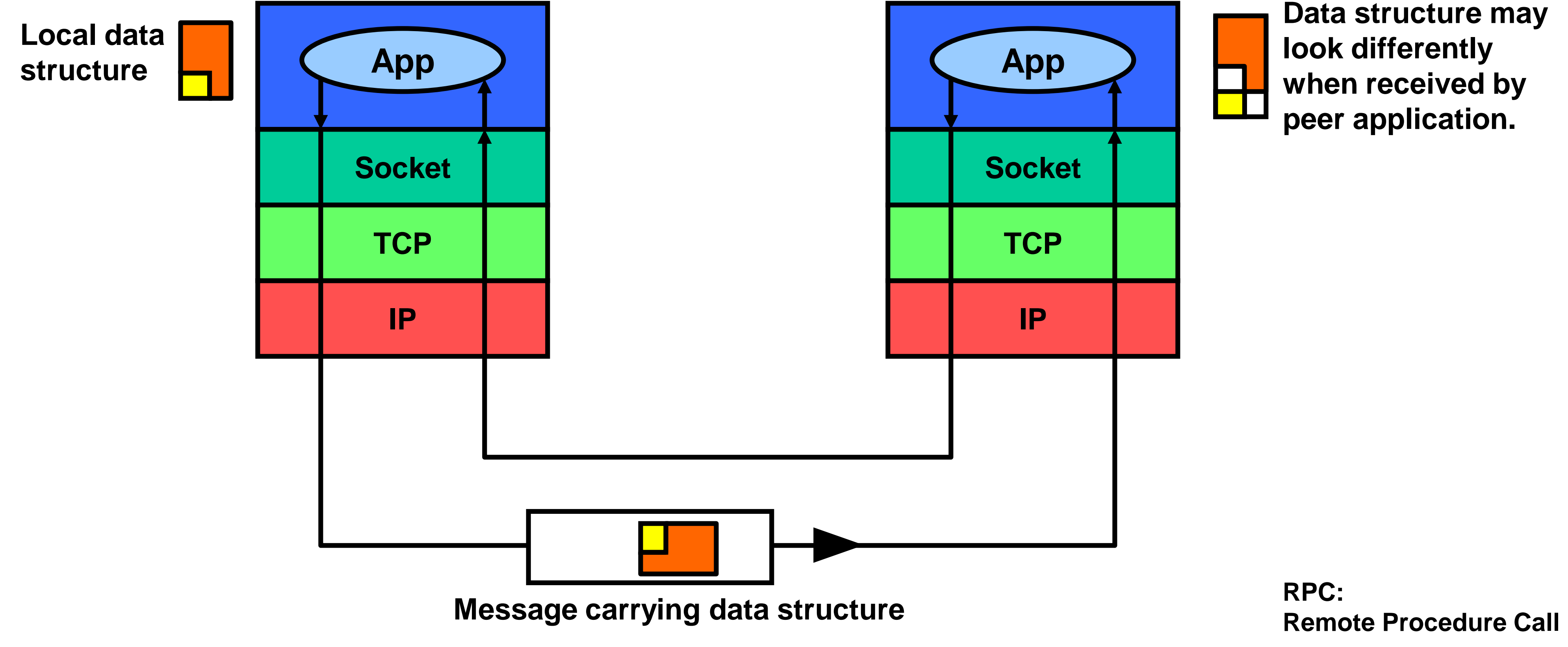
```
Socket sock = new Socket(destHostIP,destHostPort);
while not (rx = EOF) {
  rx = sock.read();
}
sock.write("I'm done");
sock.close
```

© Peter R. Egli 2017

9. Parameter marshalling / RPC transparency (1/4)

Problem:

Different implementations (C/Java, processor architecture, compiler) have different representations of data. A local data structure sent by application on host 1 may look differently to application on host 2.



9. Parameter marshalling / RPC transparency (2/4)

LSByte Least Significant Byte
MSByte Most Significant Byte

➔ E.g. Endianness:

Endianness is the ordering of bytes of a multibyte data type (integer, long integer etc.).
Network order is the way bytes (and bits) go out to the network. Network order is big endian (MSByte first).

```
//the following integer is represented differently on different
//processor architectures / operating systems
int i = 14;
```

Memory address →

n	n+1	n+2	n+3	
0 (MSByte)	0	0	14 (LSByte)	int value ‚14' in C/C++ on 32Bit big endian machine

n	n+1	n+2	n+3	
14 (LSByte)	0	0	0 (MSByte)	int value ‚14' in C/C++ on 32bit little endian machine

n	n+1	n+2	n+3	
0 (MSByte)	0	0	14 (LSByte)	int value ‚14' in Java on 32Bit/64Bit big/little endian machine

n	n+1	n+2	n+3	
0 (MSByte)	0	0	0	int value ‚14' in C/C++ on 64bit big endian machine
n+4	**n+5**	**n+6**	**n+7**	
0	0	0	14 (LSByte)	

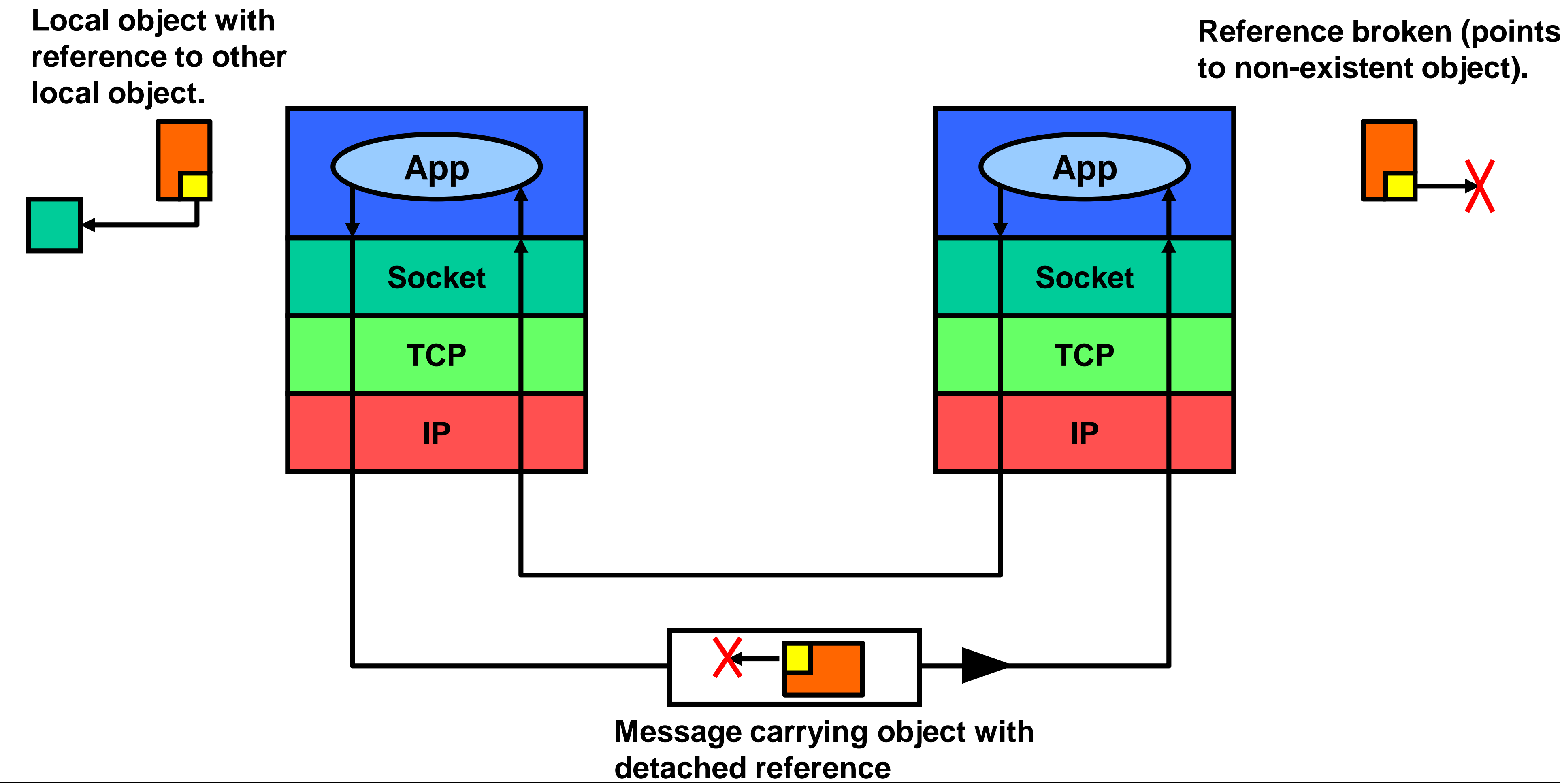
n	n+1	n+2	n+3	
14				int value ‚14' in C/C++ on 8Bit big/little endian machine

© Peter R. Egli 2017

9. Parameter marshalling / RPC transparency (3/4)

➔ **E.g. complex data structures with references:**

Complex data structures contain references to to other data structures or objects.
Such references make only sense on the local machine but not on another host.

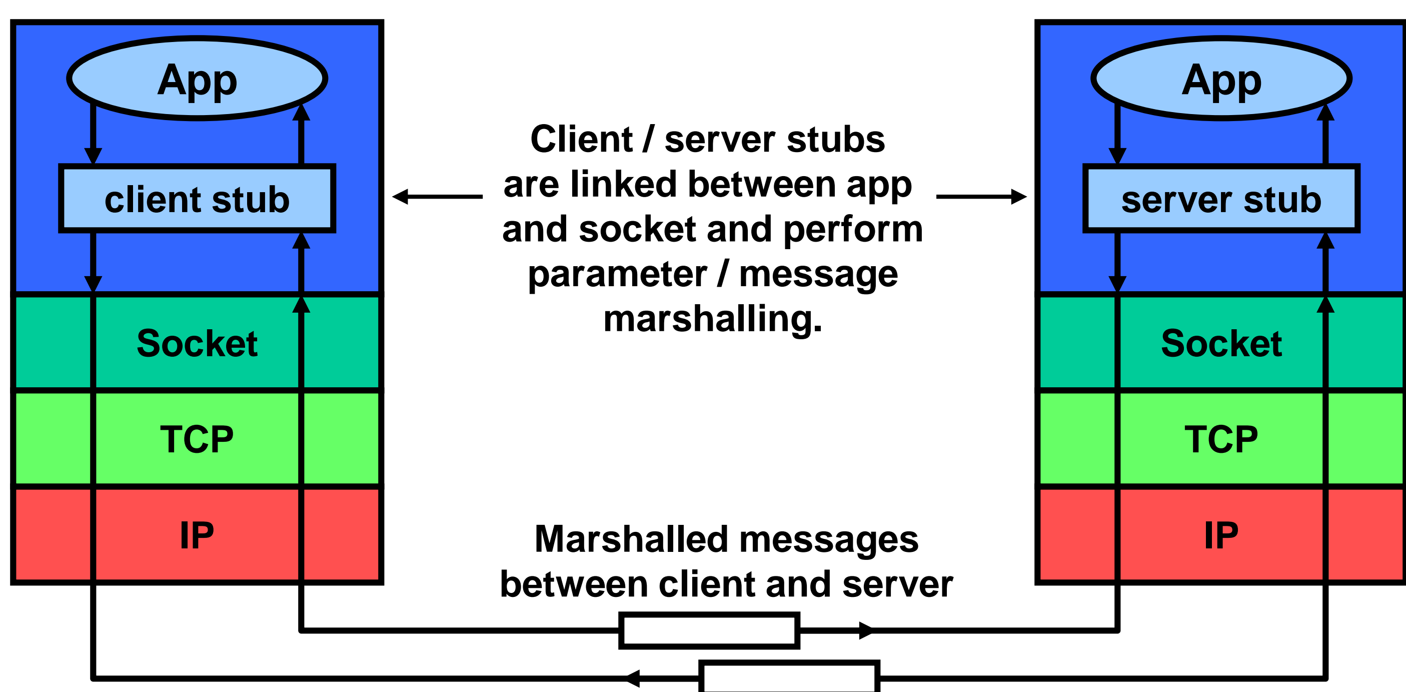


9. Parameter marshalling / RPC transparency (4/4)

➔ **Solution:**

When sending parameters over the network it is mandatory to bring them into a standard ‚canonical' format. This is called parameter marshalling (or serialization).

Stubs on the client and server marshal parameters into a standard format and vice versa.



➔ **E.g. IDL/CORBA, Interface Description Language, generates client & server stubs from abstract interface description. The stubs are then compiled by compiler together with application code.**

© Peter R. Egli 2017

10. Low level socket programming (1/2)

➔ Socket Options (SO):

Socket options allow modifying the behavior of sockets.
Generally such options should be used with caution as this makes applications dependent on the underlying socket layer (violation of layering principle).

Java (1.6) socket option support:

socket.setSoLinger(boolean on, int linger)	SO_LINGER: Define time that socket remains active to send unsent data after close() has been called (send data in transmit buffer).
socket.setSoTimeout(int timeout)	SO_TIMEOUT: Specify a timeout on blocking socket operations (don't block forever).
socket.setTcpNoDelay(boolean on)	SO_NODELAY: Enable/disable Nagle's algorithm.
socket.setKeepAlive(boolean on)	SO_KEEPALIVE: Enable/disable TCP keepalive timer mechanism.
socket.setReceivedBufferSize(int size)	SO_RCVBUF: Set the size of the receive buffer.
socket.setSendBufferSize(int size)	SO_SNDBUF: Set the size of the send buffer.
socket.setReuseAddress(boolean on)	SO_REUSEADDR: Enable reuse of port number and IP address so that after a restart an application can continue using open connections.

C/C++ socket option support:

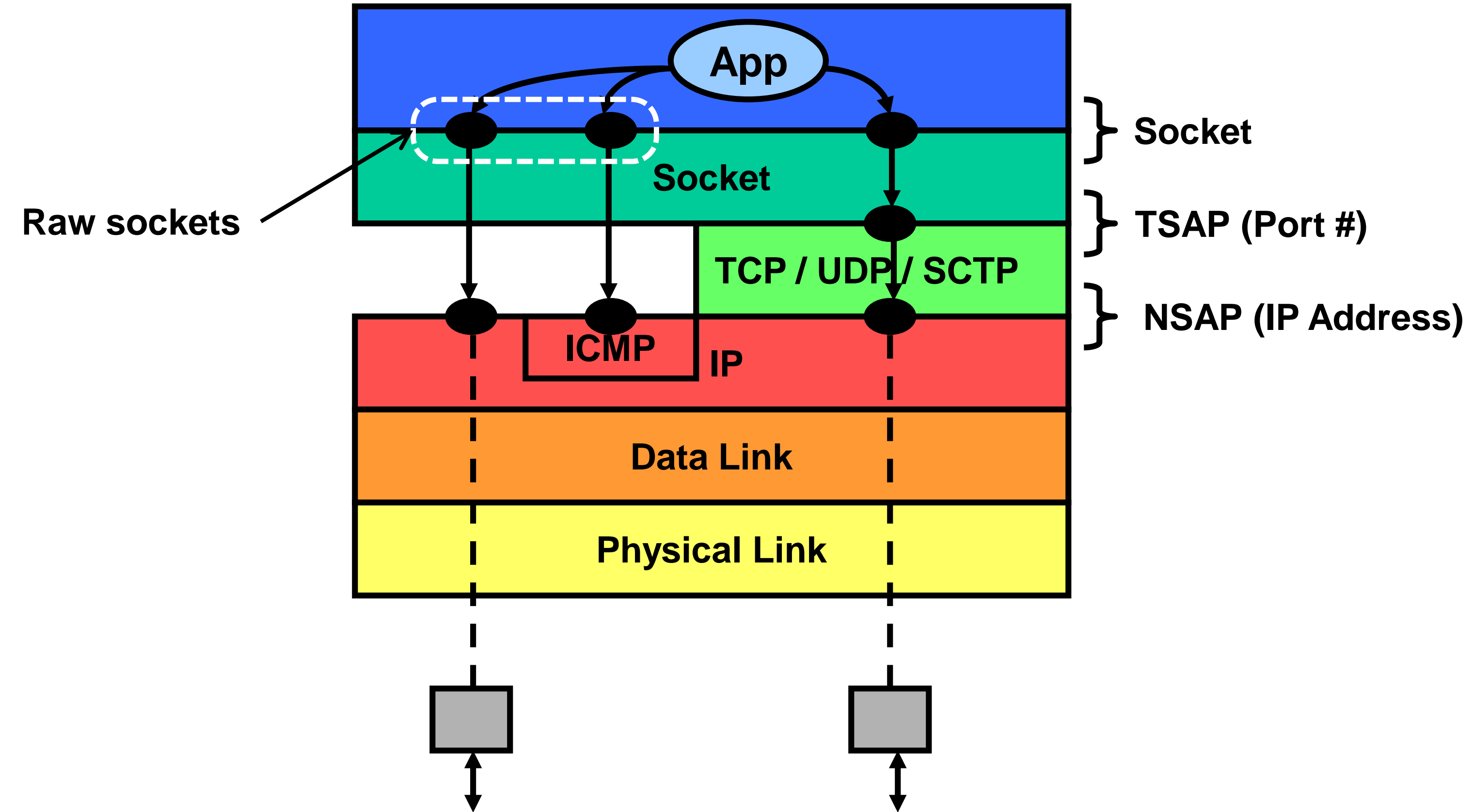
In C/C++ many more socket options can be set through setsockopt() and getsockopt() socket API calls.

© Peter R. Egli 2017

10. Low level socket programming (2/2)

➔ **Socket raw interfaces:**

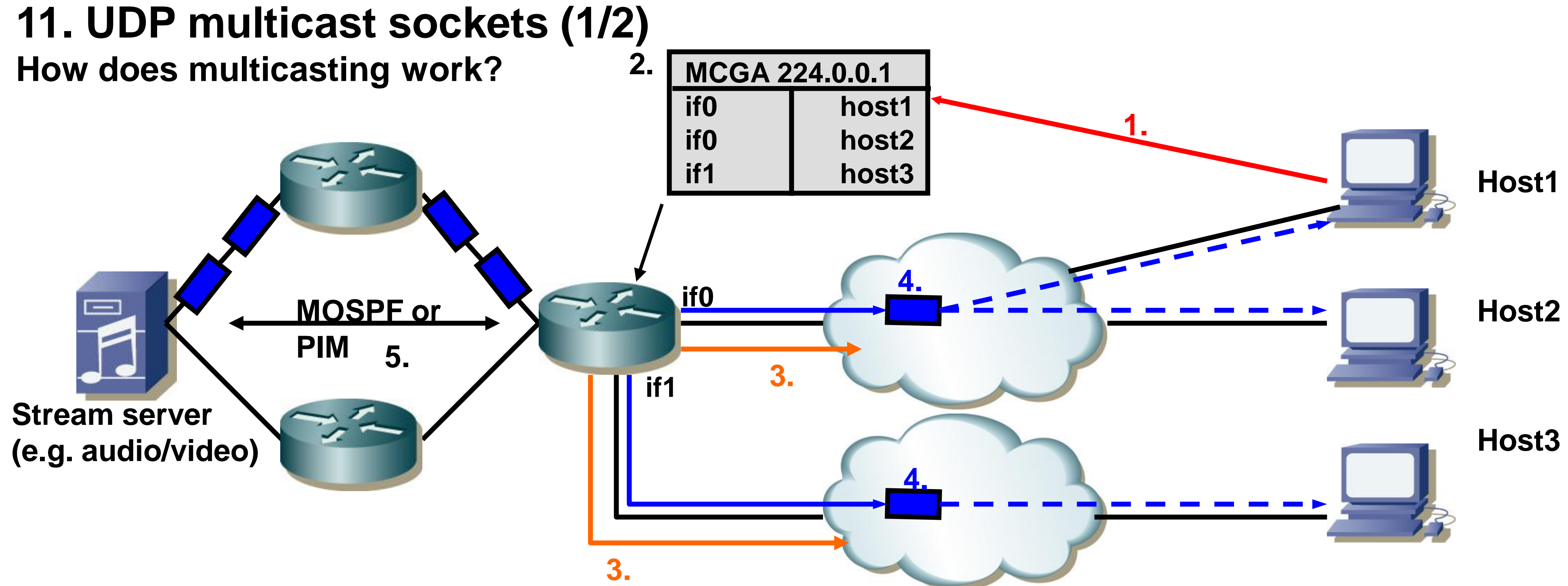
A raw socket is directly attached to the network layer without a transport layer (no TCP, UDP or SCTP layer).
This allows direct access to ICMP (e.g. for traceroute), or IP (e.g. for IPSec).
The raw interface is not available in Java due to security concerns (access to raw interface requires root access rights since the network stack runs in the kernel space).



© Peter R. Egli 2017

11. UDP multicast sockets (1/2)

How does multicasting work?



1. Hosts join multicast groups by sending IGMP (Internet Group Management Protocol) membership reports (on multicast address of interest, e.g. 224.0.1.1).
2. Multicast routers keep a table to know on which interface multicast packets are to be sent.
3. Multicast routers send periodic IGMP queries to the multicast hosts to check if they are still member of the multicast group (again sent on multicast address of interest, e.g. 224.0.1.1).
4. Upon reception of a multicast packet the multicast router performs a lookup (multicast group table with multicast group addresses MCGA) and sends the packet to all interfaces that have multicast hosts attached. The packet is sent using the corresponding multicast link address and is thus received by all multicast hosts.
5. The best (and only) route through the network (no loops etc.) is established with multicast routing protocols such as MOSPF (Multicast OSPF), PIM (Protocol Independent Multicast) etc.

11. UDP multicast sockets (2/2)

➔ Multicast is only supported on UDP (TCP is connection-oriented and thus not suitable for multicast).

➔ Multicast addresses:
Multicast addresses are class D IP addresses in the range 224.0.0.0 to 239.255.255.255.
For example:

224.0.0.9	RIP Version 2
224.0.1.1	Network Time Protocol (NTP)
224.0.0.5	All MOSPF routers

➔ Java multicast socket class:
Class MulticastSocket

MulticastSocket(int port)	Creates a multicast socket on specified port.
joinGroup(InetAddress mcastaddr)	Join a multicast group.
leaveGroup(InetAddress mcastaddr)	Leaves a multicast group (no IGMP report sent, only for hosts internal bookkeeping).
send() and receive()	Inherited methods from DatagramSocket class.

© Peter R. Egli 2017

12. TCP server socket: C/C++ versus Java example

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


int main()
{
    struct sockaddr_in serv, cli;
    char   request[REQUEST], reply[REPLY];
    int    listenfd, sockfd, n, clilen;
    if ((listenfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
        err_sys("socket error");
    memset($serv, sizeof(serv), 0);
    serv.sin_family = AF_INET;
    serv.sin_addr.s_addr = htonl(INADDR_ANY);
    serv.sin_port = htons(TCP_SERV_PORT);
    if (bind(listenfd, (SA) &serv, sizeof(serv)) < 0)
        err_sys("bind error");
    if (listen(listenfd, SOMAXCONN) < 0)
        err_sys("listen error");

    for (;;) {
        clilen = sizeof(cli);
        if ((sockfd = accept(listenfd, (SA) &cli, &clilen)) < 0)
            err_sys("accept error");
        if ((n = read_stream(sockfd, request, REQEST)) < 0)
            err_sys("read error");
        // n Bytes in request[] verarbeiten, reply[] erzeugen
        if (write(sockfd, reply, REPLY) != REPLY)
            err_sys("write error");
        close(sockfd);
    }
}
```

```java
import java.net.*;
import java.io.*;

public static void main(String[] args)
{
    ServerSocket serv;
    Socket       cli;
    PrintStream  out;
    InputStream  in;

    try {
        serv = new ServerSocket(33333);
    } catch(IOException e) { ... }




    while(true) {
        try {
            cli = serv.accept();
        } catch(IOException e) { ... }
        try {
            out = cli.getOutputStream();
            in = cli.getInputStream();
            String request = in.readln();

            // reply erzeugen...

            out.println(reply);
            cli.close();
        } catch (IOException e) { ... }
    }
    try {
        serv.close();
    } catch (IOException e) { ... }
}
```

© Peter R. Egli 2017

13. TCP client socket: C/C++ versus Java example

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

int main(int argc, char *argv[])
{
    struct sockaddr_in serv;
    char   request[REQUEST], reply[REPLY];
    int    sockfd, n;

    // Prüfen der Parameter...

    memset(&serv, sizeof(serv), 0);
    serv.sin_family = AF_INET;
    serv.sin_addr.s_addr = inet_addr(argv[1]);
    serv.sin_port = htons(TCP_SERV_PORT);
    if (connect(sockfd, (SA) &serv, sizeof(serv)) < 0
        err_sys("connect error");

    // request[] initialisieren...

    if (write(sockfd, request, REQUEST) != REQUEST)
        err_sys("write error");
    if (shutdown(sockfd, 1) < 0)
        err_sys("shutdown error");
    if ((n = read_stream(sockfd, reply, REPLY)) < 0)
        err_sys("read error");

    // n Bytes von reply[] verarbeiten...

    exit(0);
}
```

```
import java.net.*;
import java.io.*;

public static void main(String[] args)
{
    Socket       clnt;
    PrintStream  out;
    InputStream  in;

    try {
        clnt = new Socket("localhost", 33333);
    } catch(IOException e) { ... }

    try {
        out = clnt.getOutputStream();
        in = clnt.getInputStream();
        out.print("Hallo Server!");
        String reply = in.readln();
        clnt.close();
    } catch (IOException e) { ... }
}
```

© Peter R. Egli 2017

14. IPv6 sockets (1/2):

Most host platforms (Linux, Windows, Sun) already support IPv6.

➔ **IPv6 sockets with Java:**
- → Java supports IPv6 since version 1.4.
- → No difference to IPv4 sockets.
- → IPv6 automatically enabled when detected.
- → No source code change, no bytecode change required for IPv6 as long as the application does not use numeric IP addresses.

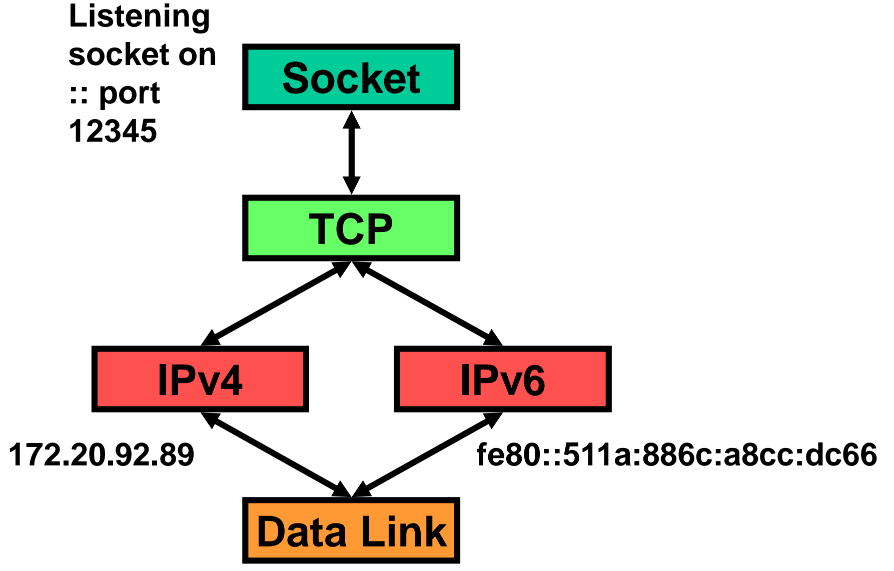
➔ **Java IPv6 API:**

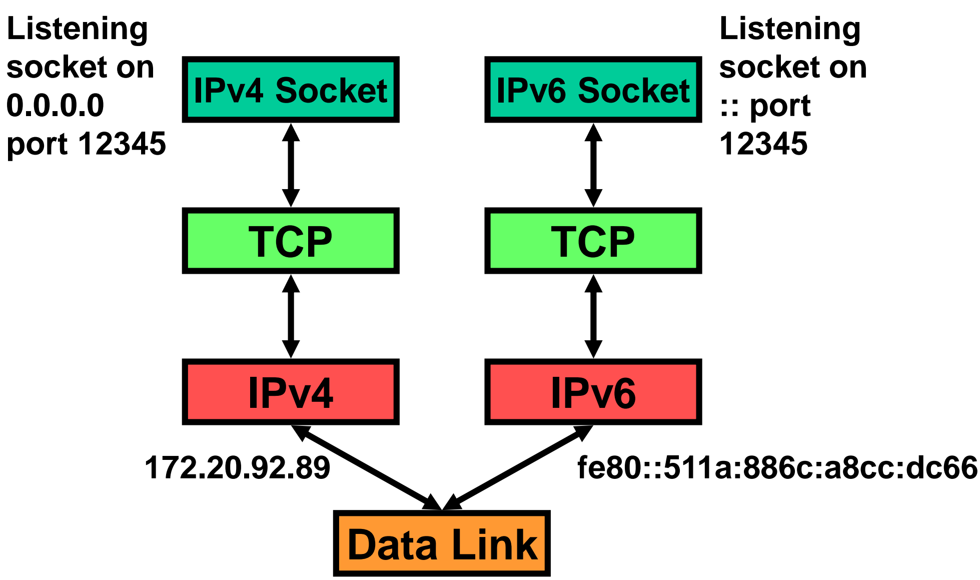
java.net.Inet4Address	IPv4 address class
java.net.Inet6Address	IPv6 address class
java.net.preferIPv4Stack	Property to set preference for IPv4.
java.net.preferIPv6Addresses	Property to set preference for IPv6.

N.B.: The properties are only accepted as VM arguments on startup of a program. They can not be changed at runtime.
Example: -Djava.net.preferIPv4Stack=false -Djava.net.preferIPv6Stack=true

14. IPv6 sockets (2/2):

Scenarios:





The listening socket accepts connections to 172.20.92.89 and fe80::511a:886c:a8cc:dc66 on port 12345.

Windows is dual stack since Windows Vista.

IP4 socket accepts connections only on 172.20.92.89.

IPv6 socket accepts connections only on fe80::511a:886c:a8cc:dc66.

© Peter R. Egli 2017