TCP/IP – part 2
(BSD UNIX) sockets

Last modification date: 08.04.2020

Standards

- Socket interface was developed in the University of California, Berkeley and became a part of 4.2BSD UNIX in 1983.
- Sockets in Portable Operating System Interface (POSIX) standards,
  - The first version: IEEE Std. 1003.1-1988, updated in 1990 (also as ISO/IEC 9945-1:1990)
  - Recent versions: IEEE Std. 1003.1-2008 , Issue 7 (Technical corrigendum 1 in 2013)
    - Std. 1003.1B in chapter 2.10 describes socket interface, chapter 3 gives description of socket API functions.
- Open Group Technical Standard: Base Specifications, Issue 6, December 2001.
- Common standard IEEE 1003.1™ POSIX ® and The Open Group Base Specifications Issue 6 (2008).
  - IEEE and The Open Group agree to include material of the standards into the projects: Linux Man Pages and FreeBSD
- Windows Sockets API (WinSock) – a variant of socket interface defined by Microsoft; it is partially compatible with UNIX/POSIX (see: homepage, Winsock Reference)

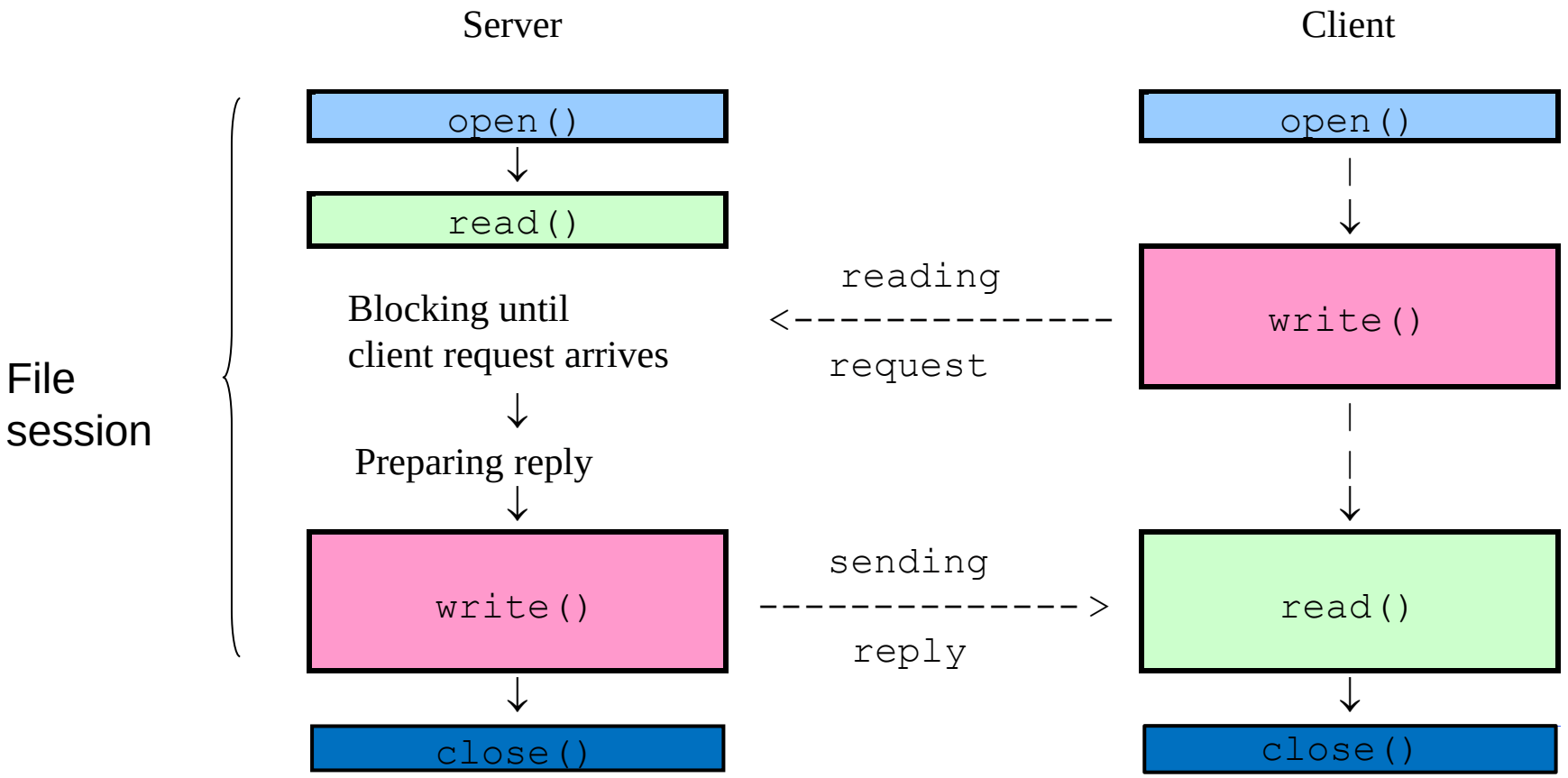
Basic features of BSD UNIX sockets

- A socket is an endpoint for communication. It has a specific type and is associated with a specific protocol. A socket is accessed via a descriptor obtained when the socket is created.
- Unix sockets are integrated with I/O subsystem
  - Socket descriptors belong to the same space as file or FIFO descriptors.
  - It is possible to use read/write functions for communication (between connected transport endpoints)
  - Socket functionality is provided by kernel and so it is available via system function calls.
  - Error handling (API functions error code, **errno** variables, error messages) is provided by operating system (as for other low-level I/O functions)
- Unix domain (local) sockets are visible in the file system as special files of type **socket**; they are used for inter-process communication within the same system.

Example: /tmp/.X11-unix/X0 entry is typically used by Linux system for X Window System related communication.

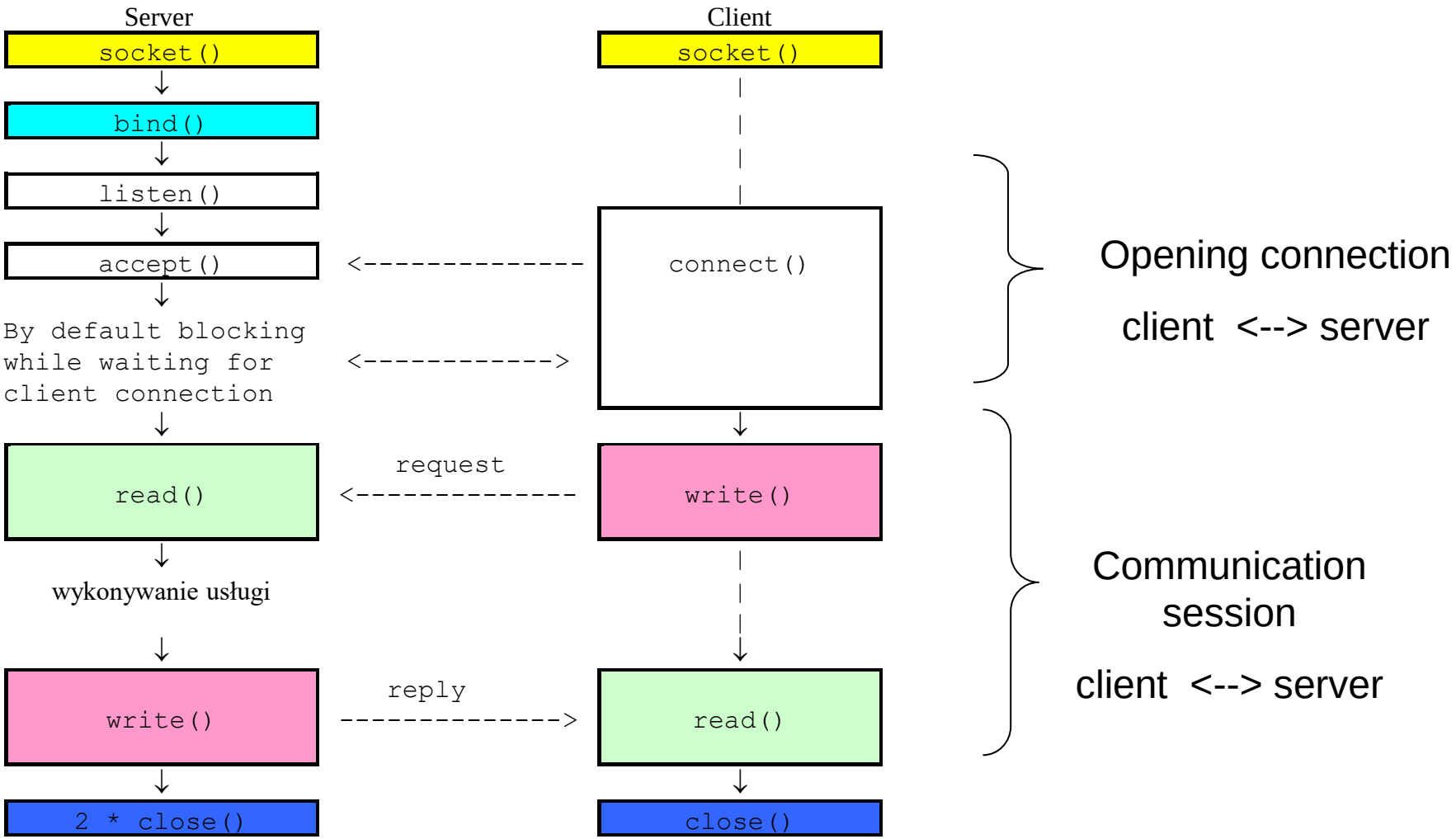
Simple communication (request-reply)

Communication via FIFOs or files



Simple communication (request-reply) with TCP

Connection-oriented communication(TCP)



Socket creation

```
#include <sys/socket.h>
int socket(
  int family,  /* protocol family
        POSIX: protocol families supported by the system are implementation-defined,
        typical: PF_INET, PF_INET6, PF_LOCAL (PF_UNIX),  PF_IPX, PF_PACKET */
  int type,  /* type of socket to be created
        POSIX: SOCK_STREAM, SOCK_DGRAM,  SOCK_SEQPACKET and
        implementation-defined; typical: SOCK_RAW, SOCK_PACKET */
  int protocol  /* protocol selection, if 0 – default protocol for the default protocol for
                   this address family and type shall be used */
);
```

- **socket()** creates an endpoint of communication; returns non-negative socket descriptor on success, otherwise -1 after setting global variable **errno** (see **man errno**) to contain error code (see **man socket**).
- Initial socket state: CLOSED.
- The process may need to have appropriate privileges to use the **socket( )** function or to create some sockets.
- Despite initial plans each protocol family is associated with one addressing family. In consequence symbolic designators of the two families are equal, i.e.

AF_XXX == PF_XXX

Socket options

- There are a number of socket options which either specialize the behavior of a socket or provide useful information. These options may be set at different protocol levels and are always present at the uppermost "socket" level (SOL_SOCKET).
- Socket options are manipulated by two functions, **getsockopt**( ) and **setsockopt**( ):

```
#include <sys/socket.h>
 int getsockopt(int sockfd, int level, int optname,
     void *optval, socklen_t *optlen);

 int setsockopt(int sockfd, int level, int optname,
     const void *optval, socklen_t optlen);
```

level: determines API level of the option (eg. SOL_SOCKET)
optval and **optlen** are used to access option values for **setsockopt**(). For **getsockopt**() they identify a buffer in which the value for the requested option(s) are to be returned (for specified socket **sockfd**)
Note: **fcntl**() and **ioctl**() calls can also modify some socket properties (e.g. non-blocking mode).

Socket options (SOL_SOCKET)

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).

TCP socket options (IPPROTO_TCP)

Option	Parameter type	Parameter meaning
TCP_KEEPALIVE	int	# of secs between keep alive messages
TCP_MAXRT	int	Max. retransmission time
TCP_MAXSEG	int	Max. size of TCP segments
TCP_NODELAY	int	avoid coalescing of small segments (turn-off Nagle algorithm)

Linux:

TCP_QUICKACK	int	Enable/disable quick ACK mode. When enabled acknowledgments are sent immediately, and not delayed (as usual). Since Linux 2.4.4

Address structures (POSIX)

- Generic socket addressing structure (sys/socket.h)

```
struct sockaddr {/* Addressing structure; POSIX */
  sa_family_t   sa_family;      /* Address family: AF_xxx */
  char          sa_data[];      /* Socket address (variable-length data) */
}
```

- Addressing structure (**sockaddr_in**) for IPv4 protocol (netinet/in.h)

```
typedef uint32_t        in_addr_t;
typedef uint16_t        in_port_t;

struct in_addr { in_addr_t s_addr; }; /* 32-bit IPv4 address */

struct sockaddr_in{ /* IPv4 addressing structure; it has to contain the following
  fields*/
  sa_family_t   sin_family;     /* AF_INET */
  in_port_t     sin_port;       /* 16b. port number(uint16_t) */
  struct in_addr sin_addr;      /* 32b. IPv4 address (uint32_t) */
};

/* Note: sin_port and sin_addr are in network order */
```

Address structures – cont.

- Addressing structure (**sockaddr_in6**) for IPv6 protocol (netinet/in.h)

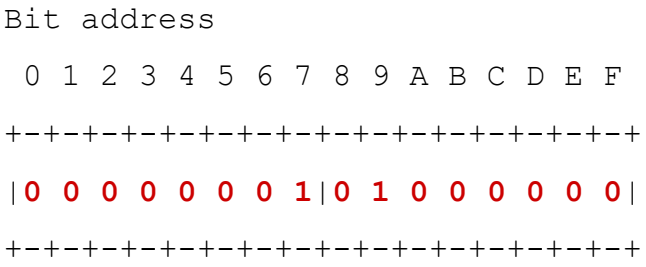
```
struct in6_addr  {uint8_t s6_addr[16];}; /* 128-bit IPv6 address */

struct sockaddr_in6; /* Addressing structure for IPv6. It has to have the
  following fields: */

  sa_family_t     sin6_family;     /* AF_INET6 */

  in_port_t       sin6_port[14];   /* 16b. Port number (uint16_t) */

  uint32_t        sin6_flowinfo;   /* IPv6 traffic class and flow information */

  struct in6_addr sin6_addr;       /* IPv6 address */

  uint32_t sin6_scope_id;          /* Set of interfaces for a scope */

/* Note: sin6_port and sin6_addr are in network order */
```

- Addressing structure (**sockaddr_un**) for UNIX domain (AF_UNIX) (sys/un.h)

```
typedef uint32_t in_addr_t;

struct in_addr   in_addr_t s_addr; /* 32-bitowy adres IPv4 */
struct sockaddr_un{ /* Addressing structure for UNIX domain.
                       It has to have the following fields: */

  sa_family_t     sun_family;      /* AF_UNIX / AF_LOCAL */
  char            sun_path[];      /* path name, POSIX: undefined length
                  typical length: 92 do 108 ; note: no port number */

};
```

Byte order conversion function

- **Network order** of bytes in numbers assumes that the first byte is the most important/weighted (*big-endian*). E.g. the single-byte integer **320** is represented as follows --->



- Byte ordering can be changed with conversion functions: **htons**, **htonl**, **ntohs**, **ntohl**

Host order	Conversion function	Network order
uint16_t	-------> **htons** ------>	**uint_16_t**
uint16_t	<------- **ntohs** <------	**uint_16_t**
uint32_t	-------> **htonl** ------>	**uint32_t**
uint32_t	<------- **ntohl** <------	**uint32_t**

Address conversions

- IPv4 address conversion without error handling (POSIX, but use not recommended)

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

- `in_addr_t inet_addr(const char *p);` /* converts the string pointed to by *p*, in the standard IPv4 dotted decimal notation, to an integer value suitable for use as an Internet address; on error it returns (**in_addr_t**)(−1) */
- `char *inet_ntoa(struct in_addr inaddr);` /* convert the Internet host address specified by *in* to a string in the Internet standard dot notation; the function returns pointer to the buffer with network address. */

- IPv4 address conversion with error handling (non-POSIX)

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

- `int_t inet_aton(const char *p, struct in_addr *addrptr);` /* converts the string pointed to by *p*, in the standard IPv4 dotted decimal notation, to an integer value (stored at **addrptr** address) suitable for use as an Internet address; the function returns 1 on success and 0 otherwise (errno is not set) */

Address conversions- cont.

- IPv4 and IPv6 address conversion with error handling (POSIX)

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

const char *inet_ntop( /* convert a numeric address (IPv4/v6) into a text string suitable for
          presentation */
          int af, /* address family*/
          const void *restrict src, /* a ptr to a buffer with address (IPv4 if af==AF_INET */
          char *restrict dst, /* a ptr to a buffer to receive resulting text string */
          socklen_t size /* size of the buffer pointed at by dst */
); /* Upon success dst is returned, NULL for error (error code in errno). See man inet_ntop. */
```

```
#include <arpa/inet.h>
int inet_pton(
/* converts an address in its standard text presentation form into its numeric binary form. */
   int af, /* address family*/
   const char *restrict src, /* a ptr to to the string being passed in.*/
   void *restrict dst /* a ptr to a buffer to receive addres (in network order). Size of the buffer
                                 32b for af==AF_INET, 128b for af==AF_INET6 */
);
```

- For IPv4 the dot format (**ddd.ddd.ddd.ddd**) is assumed for IPv6 **x:x:x:x:x:x:x:x** format (see also **man inet_pton).**

- On success **1** is returned; it shall return 0 if the input address is not valid or −1 with **errno** set to [EAFNOSUPPORT] if the **af** argument is unknown.

Using host names

- Structure with host addresses(**<netdb.h>**)

```
struct hostent {
char *   h_name;/* official host name */
char **  h_aliases; /* array of alias names, NULL terminated */
int      h_addrtupe; /* address type (AF_INET or AF_INET6) */
int      h_length; /* address length in bytes (4 or 16) */
char **  h_addr_list; /* array of pointers to (text) addresses
                         (IPv4 and IPv6); terminated with NULL */
};
#define h_addr   h_addr_list[0] /* the first address in the array */
```

- Retrieval of host address from a database: `gethostent(),endhostent()`
- Host name-address conversion functions (removed in POSIX.1-2008)

```
struct hostent* gethostbyname(const char *host/* host name*/
```

);/* The function returns a pointer to a **hostent** structure with the requested address; on error NULL is returned (after setting a global variable **h_errno**, defined in <netdb.h>)

*/

```
struct host * gethostbyaddr(const char *address,  /* a ptr to a binary IP address
                                                     (struct in_addr, or struct in6_addr) */
   int len,  /* address length*/
   int typ  /* address type (e.g. AF_INET) */
);  /* for return values see gethostbyname() description above */
```

Example of gethostbyname() use

```
int main(int argc, char *argv[]){
  char *host;
  int   sockfd, port=….;
  struct sockaddr_in   serv_addr;
  if ( argc>=2){
        host=argv[1];
        if ( argc==3 ) port=atoi(argv[2]);
  } else   host=”127.0.0.1”;
  fprintf(stderr,"%s: server %s, TCP port %d\n", argv[0],host,port);
  memset(&serv_addr,0, sizeof(serv_addr));
  serv_addr.sin_family  = AF_INET;
  if (!isdigit(host[0]) || inet_aton(host,&serv_addr .sin_addr)==0){
  struct hostent *hp=gethostbyname(host);
        if(hp==NULL){
           fprintf(stderr,"%s: host %s not found\n",argv[0],host);
               exit(1);
        }
        serv_addr.sin_addr.s_addr=
               ((struct in_addr *)(hp->h_addr))->s_addr;
  }
  serv_addr.sin_port= htons(port);
..................
}
```

Address conversions in POSIX

Network address and service translation

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
/* Given node and service, which identify an Internet host and a  service,
   getaddrinfo(2)  returns  one  or more addrinfo structures, each of which
   contains an Internet address that can be specified in a call to bind(2)
   or  connect(2). */
int getaddrinfo (const char *node, const char *service,/* one can be NULL */
                 const struct addrinfo *hints,/* selection criteria */
                       struct addrinfo **res);/* ptr to a linked list of
                                                 addrinfo structures */
void freeaddrinfo(struct addrinfo *res);/* memory dealloc. of struct addrinfo*/
const char *gai_strerror(int errcode);/*  get*info() error code -> string */
```

Address to name translation in protocol independent manner

```
int getnameinfo (
  const struct sockaddr *sa, socklen_t salen,/* socket addr */
  char *host, socklen_t hostlen, /* caller-alloc. buffer to receive host name */
  char *serv, socklen_t servlen, /* caller-alloc. buffer to receive service name
        (names are null terminated) */
  int flags /* modifies function behavior, see getnameinfo(3) */
); /* performs protocol-independent conversion: socket address to a corresponding
  host and/or service. Error codes translated to strings by gai_strerror() */
```

Address translations example (Tut.7)

```
struct sockaddr_in make_address(char *address, char *port){
        int ret;
        struct sockaddr_in addr;
        struct addrinfo *result;
        struct addrinfo hints = {};
        hints.ai_family = AF_INET;
        if((ret=getaddrinfo(address,port, &hints, &result))){
                fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(ret));
                exit(EXIT_FAILURE);
        }
        addr = *(struct sockaddr_in *)(result->ai_addr);
        freeaddrinfo(result);
        return addr;
}
```

```
Note (see getaddrinfo(3)):
struct addrinfo {
  int              ai_flags;       /* see getaddrinfo(3) */
  int              ai_family;      /* AF_INET, AF_INET6, AF_UNSPEC */
  int              ai_socktype;    /* SOCK_STREAM, SOCK_DGRAM */
  int              ai_protocol;    /* 0 - def. */
  socklen_t        ai_addrlen;      /* socket address in bytes */
  struct sockaddr *ai_addr;        /* socket address */
  char            *ai_canonname;   /* official host name */
  struct addrinfo *ai_next;        /* used for making linked list */
}
```

Binding a name (address) to a socket

```
#include <sys/socket.h>
int bind{
  int    sockfd;  /* socket descriptor */
   const struct sockaddr *addr;  /* pointer to a structure containing the address
                                    to be bound to the socket.*/
  socklen_t addrlen;  /*  the length of the sockaddr structure pointed to by addr (in bytes)  */
}
```

- **bind()** shall assign a local socket address to a socket identified by descriptor **sockfd** that has no local socket address assigned. Sockets created with the **socket**( ) function are initially unnamed; they are identified only by their address family. **bind()** returns 0 on success, and -1 otherwise (setting errno, see **man bind**). In Unix domain the socket pathname should be absolute (rather than relative). Default access rights (777) are typically affected by **umask** of the process.
- If **port==0**, the system automatically assigns an **ephemeric port**.
- If IPv4 address is equal to INADDR_ANY (or IPv4 address is equal to in6addr_any), then for a multihomed host the network software delays local address assignment until:
  - Socket becomes connected (TCP), or
  - A datagram will be sent (UDP)
- Internet host implementations use two different conceptual models for multihoming – strong and weak. In strong model datagrams are accepted only when the address in the datagram agrees with the address of the receiving interface (RFC1122).

Opening TCP connection – client side

Client side (active):

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


int connect(
  int sockfd,                                  /* client socket descriptor*/
  const struct sockaddr *servaddr,             /* ptr at the server address structure */
  socklen_t servaddrlen                        /* size of the server address structure */
);
/* The function initiates 3-phase handshake; on success it returns 0, and -1 on error (after
   setting errno) */
```

Notes:

- In Internet domain an unbound socket (sockfd) is automatically bound, using ephemeral port. In Unix domain an unbound socket will not be automatically bound to a pathname with a connect() call.
- In Unix domain an overflow of server waiting queue results in connect returning -1 with `errno=ECONNREFUSED`
- If connect() returns prematurely, due to an asynchronous signal handling, connection handling is continued “behind the scene”, and it is not possible to resume waiting for connection by calling connect again(). See Tutorial 7 to learn how to handle this situation.

Opening TCP connection – server side

Server side (passive):

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


int listen(
   int sockfd,  /* server socket (to become passive) */
   int backlog  /* size of the queue of pending connections */
);/* the function establishes a queue of pending connections,  marks the socket passive and
    enques connection indications; on success it returns 0, and -1 on error (after setting errno) */

int accept(
   int sockfd,  /* descriptor of a passive socket */
   struct sockaddr *restrict cliaddr,/* ptr to a structure which is to keep
                                        address of the recently        connected client*/
   socklen_t *restrict addrlen  /* size of the structure pointed at by cliaddr */
} /* the function makes connection to the longest waiting client; it returns 0, and -1 on error
    (after setting errno)  */
```

Opening TCP connection - remarks

- Listening socket and connected socket are associated with **the same TCP port**.
- New (connected) socket inherits options: `SO_DEBUG, SO_DONTROUTE, SO_SNDBUF, SO_KEEPALIVE, SO_LINGER, SO_OOBINLINE, SO_RCVBUF`, but no options which can be set with **`fcntl`**`()` with argument `F_SETFL` (e.g. non-blocking mode of operation)
- Asynchronous signal handling during execution of interruptible (long) blocking function calls such as **`accept`**`()` or **`connect`**`()` causes return of –1 after setting **`errno`** to **`EINTR`**.
- Default behavior of **`connect`**`()` and **`accept`**`()` (i.e. blocking until the function can be successful or when an error is detected) can be changed – by setting non-blocking mode of socket operations.

```
#include <fcntl.h>

int sockfd, val;

val=fcntl(sockfd, F_GETL, 0);

if(fcntl(sockfd, F_SETFL, val|O_NOBLOCK)) {/* error */};
```

Warning: until the mode is set back to default (blocking) all interruptible/normally blocking functions (e.g. **`read`**`()`) will be executed without blocking.

TCP sockets – data transfer

- Basic and extended data transfer functions:

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

ssize_t read(int sockfd, void *buff, size_t nbytes);

ssize_t write(int sockfd, const void *buff, size_t nbytes);

ssize_t recv(int sockfd, void *buff, size_t nbytes, int flags);

ssize_t send(int sockfd, const void *buff, size_t nbytes, int flags);
```

E.g.. `flags==MSG_PEEK` — receive operation to return data from the beginning of the receive queue without removing that data from the queue
`flags==MSG_OOB` — requests receipt of out-of-band (OOB) data that would not be received in the normal data stream or sending OOB data
`flags==MSG_DONTWAIT` — non-blocking operation
`flags==MSG_WAITALL` — on stream sockets recv() blocks til all data is retrieved.

- Read/write data into/from multiple buffers:

```
struct iovec{    /* buffer for readv/writev*/
  void *iov_base; /* starting address*/
  size_t iov_len; /* number of bytes to transfer */
};

ssize_t readv(int sockfd, const struct iovec *iov, int iovcnt);

ssize_t writev(int sockfd, const struct iovec *iov, int iovcnt);
```

Remarks on data transfer

- The most powerful data transfer functions are **`recvmsg`** and **`sendmsg`** (see **`man send, man recv`**). For UNIX domain socket they enable sending active file descriptors between processes.
- An attempt to send data via a socket which is closed for reading (on its other side) results in a failure with error code **`EPIPE`**, and **`SIGPIPE`** signal is sent to the attempting process. Note that the above happens after the sending side is notified about closed connection or timeout on previously sent segment was reached.
- If a read/write function is interrupted by signal delivery before its completion an error code **`EINTR`** is set.
- While in blocking mode a **read()** call for a socket which has the other end closed for writing results in reading unread bytes. After all bytes are read the next **read()** call returns 0.
- Sending function typically blocks until it sends out the requested number of bytes. When `n=write()` with `n>0` we can only know that `n` bytes were copied to the output buffer, and not that the bytes were received by the destination station. Data are removed from the output buffer only after the recipient acknowledged their reading.

Closing TCP connection

```
#include <sys/socket.h>
int shutdown{
  int sockfd;     /* socket descriptor*/
  int how;        /* SHUT_RD (0) - to be closed for reading */
                  /* SHUT_WR (1) - to be closed for writing */
                  /* SHUT_RDWR (2) - further receptions and transmissions
                           are to be disallowed */
}/* normally returns 0, but -1 for error condition */


#include <sys/socket.h>
int close{ int sockfd      /* socket descriptor */
};/* normally returns 0, but -1 for error condition */
```

close() disallows further receptions and transmissions. Other effects depend on the number of socket active users (links) and the value of socket option **SO_LINGER**.

Closing TCP connection – cont.

- **Linger** (in POSIX) - the period of time before terminating a connection, to allow outstanding data to be transferred.
- Socket option **SO_LINGER**, can be set with a call of **setsockopt()** using the following data structure:

```
struct linger
  int l_onoff; /* Indicates whether linger option is enabled*/
  int l_linger; /* Linger time, in seconds*/
};
```

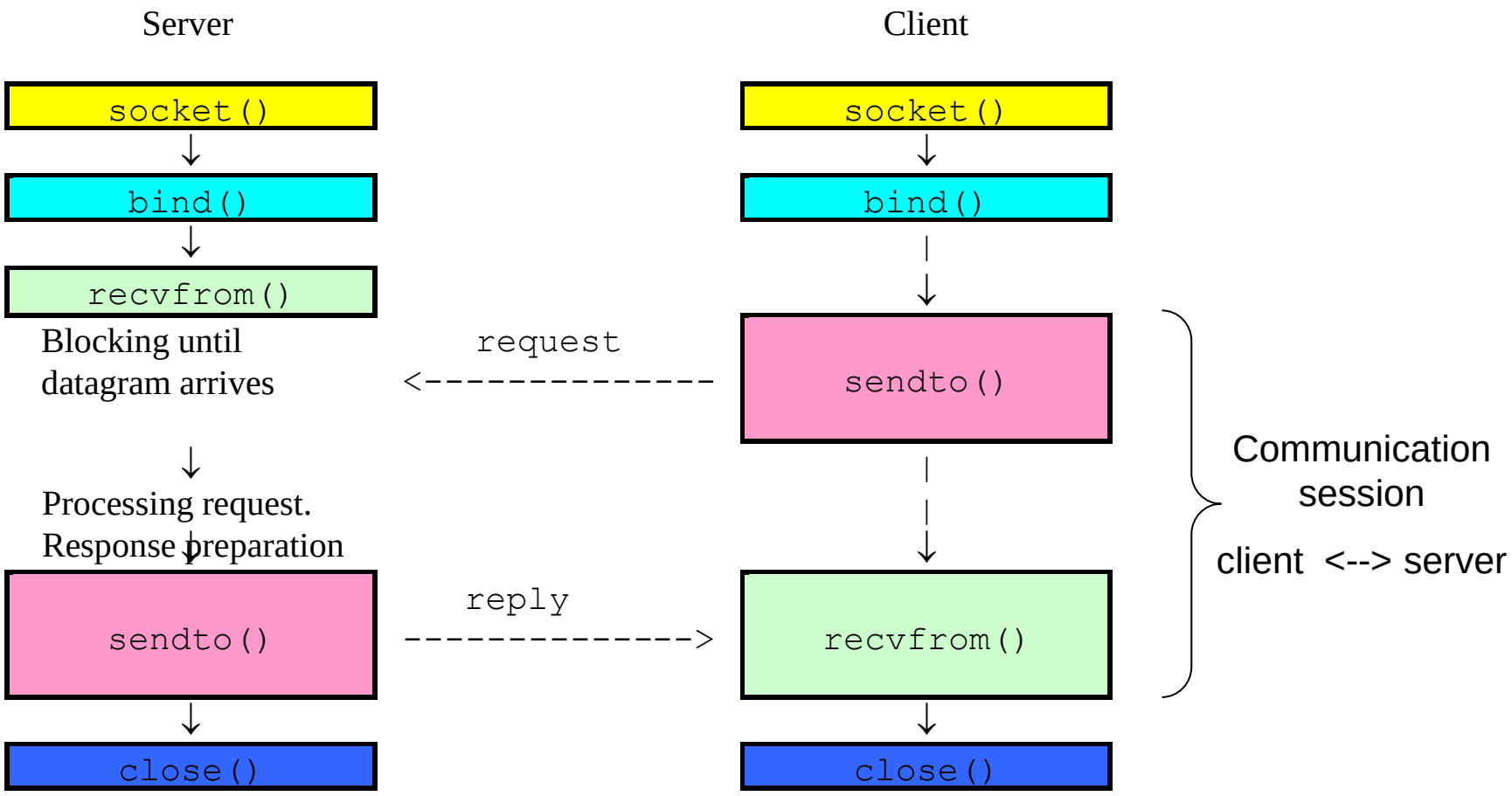
- For **l_onoff=0** a **close()** returns immediately; but the operating system attempts to deliver the content of the output buffer; the content of the reading buffer is dropped.
- For **l_onoff!=0** a **close()** or **shutdown()** call will not return until all queued messages for the socket have been successfully sent or the linger timeout (**l_linger**) has been reached. The content of the reading buffer is dropped.

SOCK_STREAM sockets – generic remarks

- The SOCK_STREAM socket type provides reliable, sequenced, full-duplex octet streams between the socket and a peer to which the socket is connected.
- A socket of type SOCK_STREAM must be in a connected state before any data may be sent or received.
- Record boundaries are not maintained; data sent on a stream socket using output operations of one size may be received using input operations of smaller or larger sizes without loss of data.
- Data may be buffered; successful return from an output function does not imply that the data has been delivered to the peer or even transmitted from the local system.
- If data cannot be successfully transmitted within a given time then the connection is considered broken, and subsequent operations shall fail. A SIGPIPE signal is raised if a thread attempts to send data on a broken stream (one that is no longer connected), except that the signal is suppressed if the MSG_NOSIGNAL flag is used in calls to **send**( ), **sendto**( ), and **sendmsg**( ).
- Support for an out-of band (OOB, urgent) data transmission facility is protocol-specific. TCP protocol guarantees 1 octet of OOB.

Simple communication (request-reply) with UDP

Connectionless (datagram) communication



UDP communication

I/O (send/receive) functions for UDP communication

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

ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
  struct sockaddr *from, socklen_t *paddrlen);/* datagram receive*/

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags, const
  struct sockaddr *to, socklen_t addrlen); /* datagram send */
```

Parameters:

sockfd – UDP socket descriptor

buff – pointer at a data buffer of length **nbytes**

from – if not NULL it points at address structure (of length stored at ***paddrlen**) to be filled with the address of datagram sender (actual size used is saved in ***paddrlen**).

to – points at address structure (of length **addrlen**) of the datagram addressee

flags – options (as for **send()** andi **recv()**)

Remarks:

- It is OK to send datagrams of length 0
- UDP socket option **SO_SNDBUF** determines maximum datagram size. An attempt to send datagram, longer than **SO_SNDBUF** sets **errno** to **EMSGSIZE**.
- A datagram has to be sent (received) in a single output (input) operation.

Connected UDP sockets

Note: `write()` does not have arguments which enable specification of a datagram addressee.

Connected UDP sockets:

- The processes which want to communicate call `connect()` specifying the other process UDP socket address as addressee.
- Data transfer can then be done using się `write()` (or `send()`) instead of `sendto()` and `read()` (or `recv()`) instead of `recvfrom()`.
- To disconnect UDP sockets one can use `connect()` call for protocol `PF_UNSPEC` (a non-implemented protocol error code should be discarded).
- Temporary connection of UDP sockets increases speed of communication for many transmission between the same two sockets.
- Socket connection enables detection of an asynchronous error. The error occurs if the datagram destination is unreachable but the sending function has already returned to the calling process (after copying data to the output buffer). Even if the receiving station sends back ICMP packet with error condition – it cannot reach the sender if connected sockets are not used. In case of many transmissions between connected UDP sockets the asynchronous error will be detected upon next datagram sent.

Retrieval of socket names (addresses)

- Name (address) of a bound socket `sockfd` can be retrieved with

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

int_t getsockname(int sockfd, struct sockaddr *name, socklen_t *namelen);
```

- Name (address) of the other side of connected socket `sockfd` can be retrieved with:

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

int getpeername(int sockfd , struct sockaddr *name, socklen_t *namelen);
```

Note:

- Before call of the above functions variable pointed at by **`namelen`** has to be initialized with length of address structure pointed at by **`name`**. Value of `*`**`namelen`** can be modified by these functions

SOCK_DGRAM sockets - generic remarks

- The SOCK_DGRAM socket type supports connectionless data transfer which is not necessarily acknowledged or reliable.
- Datagrams may be sent to the address specified (possibly multicast or broadcast) in each output operation, and incoming datagrams may be received from multiple sources. The source address of each datagram is available when receiving the datagram.
- An application may also pre-specify a peer address, in which case calls to output functions that do not specify a peer address shall send to the pre-specified peer. If a peer has been specified, only datagrams from that peer shall be received.
- A datagram must be sent in a single output operation, and must be received in a single input operation. The maximum size of a datagram is protocol-specific; with some protocols, the limit is implementation-defined.
- Output datagrams may be buffered within the system; thus, a successful return from an output function does not guarantee that a datagram is actually sent or received. However, implementations should attempt to detect any errors possible before the return of an output function, reporting any error by an unsuccessful return value.