Inter-Process communication

- http://beej.us/guide/bgnet/
- http://tldp.org/LDP/lpg/node7.html
- http://beej.us/guide/bgipc/output/html/multipage/index.html
-

System

- Composition of
  - Functions / Modules
  - Classes
  - Processes
- Processes can be running in
  - Different/same space
- Processes can be running at
  - Different/same time

Operating system infrastructure

- Operating systems/middle-wares offer
  - Execution mechanism
  - Protection Mechanisms
  - Communication mechanisms
- Protection
  - Processes are independent entities
    - One process execution does not affect other processes
    - Memory is private

Operating system infrastructure

- Nonetheless
  - Process in the same system need to exchange information or data:
    - To divide tasks
    - Increase processing power (by distributing tasks into multiple computers/processors)
    - To guarantee synchronization and consistency among them

Characteristics

- Implementation
- Scope
- Duplex
- Time-coupling
- Space-coupling
- Explicit / implicit
- Synchronization
- Process relation
- Identification
- API

Characteristics

- Implementation
  - Shared memory
  - Kernel based
    - Require data copy
      - P1 → kernel → P2
- Scope
  - Local
    - Shared memory
    - signals
  - Distributed
    - Sockets

Characteristics

- Duplex / Simplex
  -
- Time-coupling
  - Send and receiver must exist at the same time
  - Or not
- Space-coupling
  - Sender know who the receiver is
  - Or not

Characteristics

- Explicit / implicit
  - Is information transfer implicit?
- Synchronization
  - Operations are blocking?
- Process relation
  - Just father/son
  - Unrelated processes

Characteristics

- Identification
  - How are comm objects identified
    - System wide
    - Local / global
    - Int / string / files
- API
  - How are “chanels” identified in C
  - What function to read/write
  - Error handling

Pipes

- http://tldp.org/LDP/lpg/node9.html
- http://beej.us/guide/bgipc/output/html/multipage/pipes.html
-

Pipes

- M. D. McIlroy - October 11, 1964
  - We should have some ways of coupling programs like garden hose--screw in another segment when it becomes when it becomes necessary to massage data in another way This is the way of IO also.
- In Unix pipes are the original inter-process communication mechanisms

Redirect in the shell

- ls >foo
  - sends the output of the directory lister **ls** to a file named ‘foo’.
- wc < foo
  - causes the word-count utility wc(1) to take its standard input from the file ‘foo’,
  - and deliver a character/word/line count to standard output.

Pipes in the shell

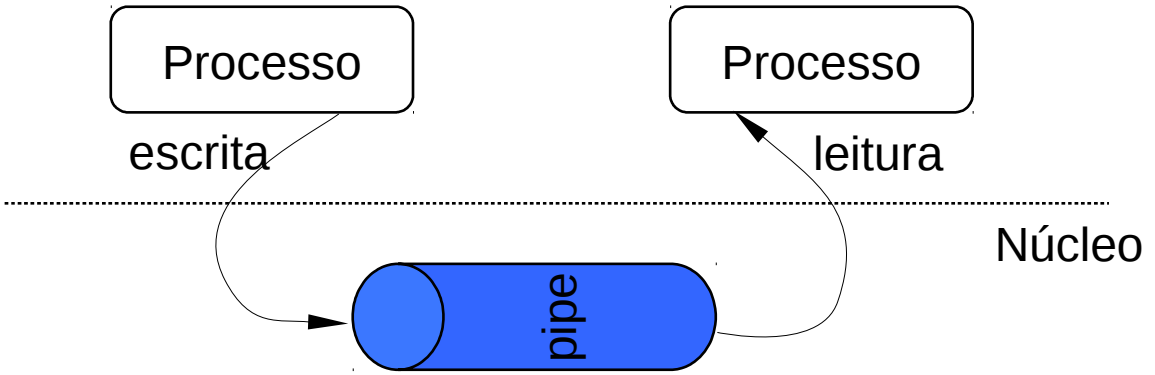
- pipe operation
  - connects the standard output of one program to the standard input of another.
  - A chain of programs connected in this way is called a pipeline.
- ls | wc
  - Counts character/word/line count for the current directory listing
- tr -c '[:alnum:]' '[\n*]' | sort -iu | grep -v '^[0-9]*$'

Pipes in the shell

- All the stages in a pipeline run concurrently.
  - Each stage waits for input on the output of the previous one,
  - no stage has to exit before the next can run.
- It is unidirectional.
  - p1 | p2 p1 → p2 p1 ← p2 
  - Impossible to pass information back
    - Just p2 dead notification
- Protocol for passing data
  - is simply the receiver's input format.

Pipes

- Read/Writes
  - File operations
- Processes
  - Should be related
    - Father/soon
    - Brothers



Pipes

- Pipe creation:
  - int pipe(int fd[2]);
  - int pipe2(int pipefd[2], int flags); /* O_NONBLOCK */
- Opens two files
  - fd[0] descriptor open for reading
  - fd[1] descriptor open for writing
- Returns
  - 0 successful
  - 1 unsuccessful (erno variable set)
- Pipes can only connect processes with a common antecessor
- Pipe information os managed as a open file

Pipes

- Communication (data read/write)
  - ssize_t read(int fd, void *buf, size_t count);
  - ssize_t write(int fd, void *buf, size_t count);
- 1st argument
  - File descriptor (fd[0] or fd[1])
- 2nd argument data buffer address (data destination/source)
- 3rd argument number of bytes to read/write
- Return number of bytes read/written
- Blocks or not (O_NONBLOCK)

Pipes

- If a process attempts to read from an empty pipe,
  - then read(2) will block until data is available.
- If a process attempts to write to a full pipe,
  - then write(2) blocks until sufficient data has been read from the pipe to allow the write to complete.
- Nonblocking I/O is possible
  - using O_NONBLOCK status flag.
- The communication channel provided by a pipe is a byte stream:
  - there is no concept of message boundaries.
-

Pipes

- Messages are limited to byte streams.
- Information flow is unidirectional
  - One process reads one process writes
  - Uses file descriptors functionality
- Major limitation
  - Processes should be related
- How to implement pipes that are accessible by other processes?
  - Giving them a name
  - Registering them in the File system

Pipes

- A pipe has a limited capacity.
  - If the pipe is full, then a write(2) will block or fail, depending on whether the O_NONBLOCK flag is set
- Different implementations have different limits for the pipe capacity.
  - Applications should not rely on a particular capacity
  - application should consume data as soon as possible
- POSIX.1-2001 says that write(2)s of less than PIPE_BUF bytes must be atomic:
  - the output data is written to the pipe as a contiguous sequence.
  - Writes of more than PIPE_BUF bytes may be nonatomic:
    - the kernel may interleave the data with data written by other processes.
  - POSIX.1-2001 requires PIPE_BUF to be at least 512 bytes.
    - On Linux PIPE_BUF is 4096 bytes.

Closing Pipes

- Closing all write ends
  - Read will return 0
- Closing all read ends
  - Write will produce SIGPIPE
- After fork processes should close not needed ends
  - For previous notifications to work

Pipes

- Implementation – Kernel / syscall
- Scope - local
- No Duplex
- Time-coupling
- Space-coupling +-
- Explicit
- Synchronization – Yes by default
- Process relation - related
- Identification - NA
- API – file operations

FIFO / Named Pipes

- http://tldp.org/LDP/lpg/node15.html
- http://beej.us/guide/bgipc/output/html/multipage/fifos.html
-

FIFO / Named Pipes

- To solve Pipes limitations
  - FIFOs were defined
    - Also referred as named pipes
- Can be used by unrelated processes
- Are referred and identified by a file in the file system
- A FIFO is special file similar to a pipe,
  - That is created in a different way
  - Instead of being an anonymous communications channel,
    - FIFO is entered into the file system by calling mkfifo()
  -

FIFO / Named Pipes

- FIFO creations
  - int mkfifo(const char *pathname, mode_t mode);
- 1st argument
  - FIFO name (full path)
- 2nd argument
  - Access permissions (like a regular file)
- Once you have created a FIFO special file in this way, any process can open it for reading or writing,
  - in the same way as an ordinary file.
- **3151348 0 prw-r--r-- 1 jnos users 0 Mar 22 10:05 test_fifo**
- On success mkfifo() returns 0.
  - In the case of an error, -1 is returned (errno is set appropriately).

FIFO / Named Pipes

- Before being used the FIFO should be opened
  - int open(const char *pathname, int flags);
- 1st argument
  - FIFO name
- 2nd argument
  - Bits that define access mode
  - O_RDONLY (just reading)
  - O_RDONLY (just writing)
  - O_NONBLOCK (non blocking I/O)
- The return value is
  - -1 in case of error
  - Or a positive file descriptor

FIFO / Named Pipes

- A FIFO has to be opened at both ends simultaneously before you can proceed to do any input or output operations on it.
  - Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa.
- Opening the FIFO in O_NONBLOCK mode
  - Returns success if other process has already opened
  - Returns -1 if it is the first open
    - Sets errno to ENXIO

FIFO / Named Pipes

- Communication (data read/write)
  - ssize_t read(int fd, void *buf, size_t count);
  - ssize_t write(int fd, void *buf, size_t count);
- 1st argument
  - File descriptor (fd[0] or fd[1])
- 2nd argument data buffer address (data destination/source)
- 3rd argument number of bytes to read/write
- Return number of bytes read/written
- Blocks or not (O_NONBLOCK)

FIFO / Named Pipes

- If a process attempts to read from an empty FIFO,
  - then read(2) will block until data is available.
- If a process attempts to write to a full FIFO,
  - then write(2) blocks until sufficient data has been read from the pipe to allow the write to complete.
- Nonblocking I/O is possible
  - using O_NONBLOCK status flag.
- The communication channel provided by a FIFO is a byte stream:
  - there is no concept of message boundaries.
- The communication is unidirectional

FIFO / Named Pipes

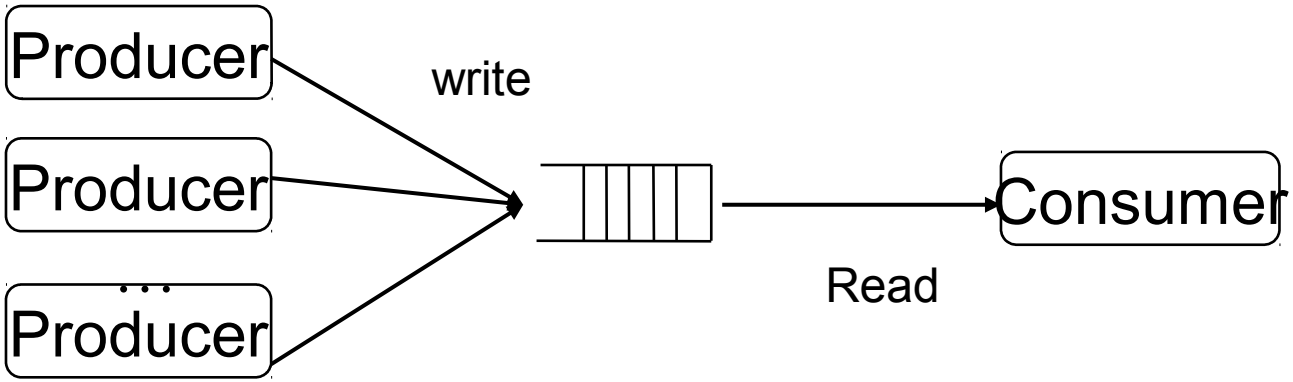
- Implementation – Kernel / syscall
- Scope - local
- No Duplex
- Time-coupling
- Space-coupling +-
- Explicit
- Synchronization – Yes by default
- Process relation - unrelated
- Identification – file name
- API – file operations

Message Queues

- SV
- POSIX

Message Queues

- Link between producer and consumer is indirect
  - Producer write messages on the queue
  - Without selecting consumer
- Consumer retrieve a message from the Queue
  - Without selecting producer
- Writings are unblocking
  - As long as resources are available
- Reading is blocking
  - When queue is empty



Message queues

- A Message Queue is a linked list of message structures
  - stored inside the kernel’s memory space and accessible by multiple processes
- Synchronization is provided automatically by the kernel
- Preallocated message buffers
- Messages with priority.
  - A message with a higher priority is always received first.
- Send and receive functions are synchronous by default.
  - Possibility to set a wait timeout to avoid nondeterminism.
- Support asynchronous delivery notifications.

Programming steps

- Define Message structure
- Create message Queue
- Connect to queue
  -
- Read messages
-
-
- Close Queue
- Destroy Queue

- Connect to queue
- Write Messages
- Close Queue

Message Structure

- Multiple processes should agree on the message structure/size
  - Application level
  - Message queues handle different sizes
- Programmer should define a structure
  - With pre-defined size
  - Able to accommodate multiple sub-types

SV MQ - Message Structure

- Example:
  - struct mymsg {
  - **long msg_type;**
  - char mytext[512]; /* rest of message */
  - int somethingelse;
  - };
- msg_type used in reception

SV MQ – creation

- msgget - get a System V message queue identifier
  - int msgget(key_t key, int msgflg);
- Create Private
  - Key – IPC_PRIVATE
  - Hinerited by chld processes
- Create Public
  - Key – not in use (ipcs)
  - Msgflag - IPC_CREAT
  - Verify if existes
    - Msgflag - O_CREAT | O_EXCL

SV MQ – opening

- msgget - get a System V message queue identifier
  - int msgget(key_t key, int msgflg);
- Open a Public MQ
  - Key – already creates MQ
  - Msgflag - NULL

SV MQ - write

- int msgsnd(int msqid,
- const void *msgp, size_t msgsz,
- int msgflg);
  - Writes a message to the queue
  - Parameters
    - Msqid – quue id (returned from msgget)
    - Message + size
    - Msgflags - IPC_NOWAIT

SV MQ - Read

- ssize_t msgrcv(int msqid,
- void *msgp, size_t msgsz,
- long msgtyp, int msgflg);
  - Reads a message from queue
  - Parameters
    - Msqid – queue id (returned from msgget)
    - Pointer to buufer + max size size
    - Type of message
    - Msgflags - IPC_NOWAIT

SV MQ - Read

- ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
  - MsgType
    - 0 – first message
    - > 0 – first message with that type
    - <0 - first message with
      - the lowest type less than or equal to the absolute value of msgtyp
  - Msgflag
    - IPC_NOWAIT
    - MSG_COPY – does nt remove message

SV MQ - destruction

- int msgctl(int msqid, int cmd, struct msqid_ds *buf);
  - Msqid
  - Cmd – IPC_RMID
  - msqid_ds NULL

POSIX MQ

POSIX MQ - Message Structure

- Array of bytes
- Priority / message selection
  - API
- Each message has an associated priority,
- Messages are always delivered to the receiving process highest priority first.
- Message priorities range
  - From 0 (low) to sysconf(_SC_MQ_PRIO_MAX) - 1 (high).
  - On Linux, sysconf(_SC_MQ_PRIO_MAX) returns 32768,
  - POSIX.1 requires a range from 0 to to 31
-

POSIX MQ – creation

- mq_open - open a message queue
  - mqd_t mq_open(const char *name,
  - int oflag, mode_t mode,
  - struct mq_attr *attr);
- Name – identifier
- Oflags -
  - O_CREAT | O_RDONLY | O_WRONLY | O_RDWR
- Mode
  - File access modes rwx / ugw 0666

POSIX MQ – creation

- mq_open - open a message queue
  - mqd_t mq_open(const char *name,
  - int oflag, mode_t mode,
  - struct mq_attr *attr);
- attr
  - NULL
  - struct mq_attr queue_attr;
    - queue_attr.mq_maxmsg = 16;
    - queue_attr.mq_msgsize = 128;

POSIX MQ – opening

- mq_open - open a message queue
  - mqd_t mq_open(const char *name, int oflag)
- Default settings
  - Name – identifier
  - Oflags -
    - O_RDONLY O_WRONLY O_RDWR

POSIX MQ - mq_open

- Creates
  - O_CREAT
- Message queue is assigned to a file
  - In /dev/msgque/
  - File name is used by other processes
- mq_close
  - close a message queue descriptor
  - Process can no longer use queue
- mq_unlink
  - removes a message queue
  - Deletes the file

POSIX MQ - write

- int mq_send(mqd_t mqdes,
- const char *msg_ptr, size_t msg_len,
- unsigned int msg_prio);
  - Writes a message to the queue
  - Parameters
    - mqdes – queue id (returned from mq_open)
    - Message + size
    - msg_priority – udes in mq_receive

POSIX MQ - read

- ssize_t mq_receive(mqd_t mqdes,
- char *msg_ptr, size_t msg_len,
- unsigned int *msg_prio);
- Reads a message from the queue
  - mqdes – queue id (returned from mq_open)
  - Message + buffer size
  - msg_priority – used in mq_receive

POSIX MQ - read

- ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,unsigned int *msg_prio);
  - Messages are always delivered to the receiving process highest priority first.
  - msg_priority –
    - NULL
    - Not NULL stores the prioryte of received message

Read/write

- Empty Queue
  - Receive Call blocks
  - mq_timedreceive
    - Block some time
- Full queue
  - Send blocks
  - mq_timedsend
    - Blocks some time
- … , const struct timespec *abs_timeout);
- Errno - ETIMEDOUT

POSIX MQ - limits

- On the user program
  - queue_attr.mq_maxmsg = 16;
  - queue_attr.mq_msgsize = 128;
- Values limited by the OS
  - /proc/sys/fs/mqueue/
- Change on:
  - /etc/security/limits.conf

Message Queues

- Implementation – Kernel / syscall
- Scope - local
- No Duplex
- Time-uncoupling
- Space-uncoupling
- Explicit
- Synchronization – Yes (reads) no (writes)
- Process relation - unrelated
- Identification – string
- API – specific API

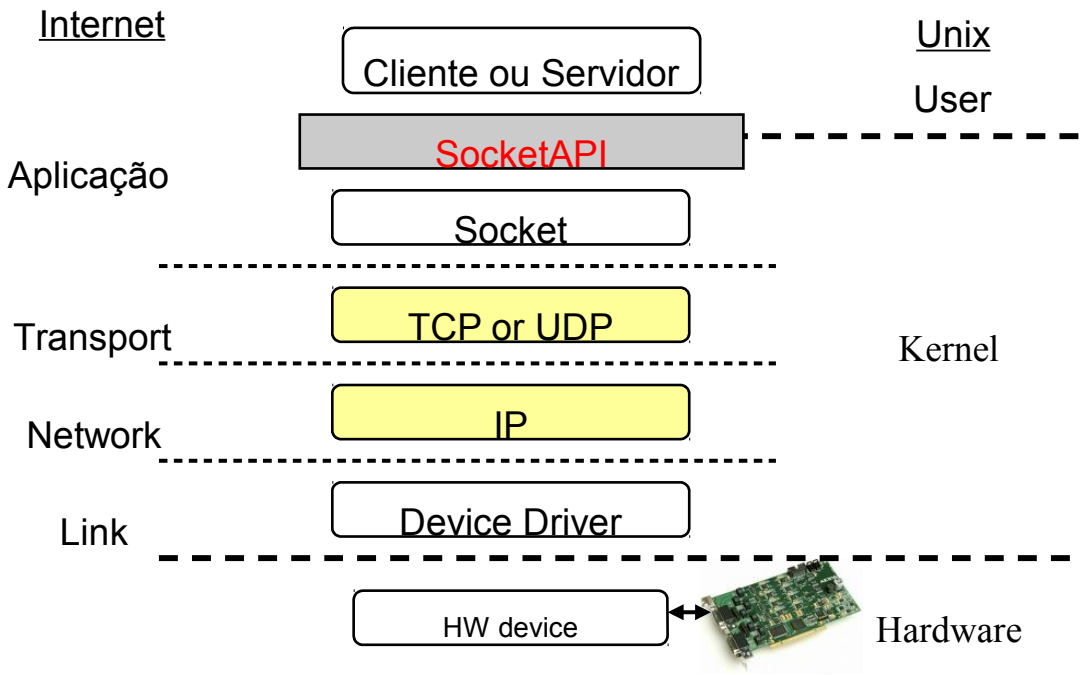
SOCKETS

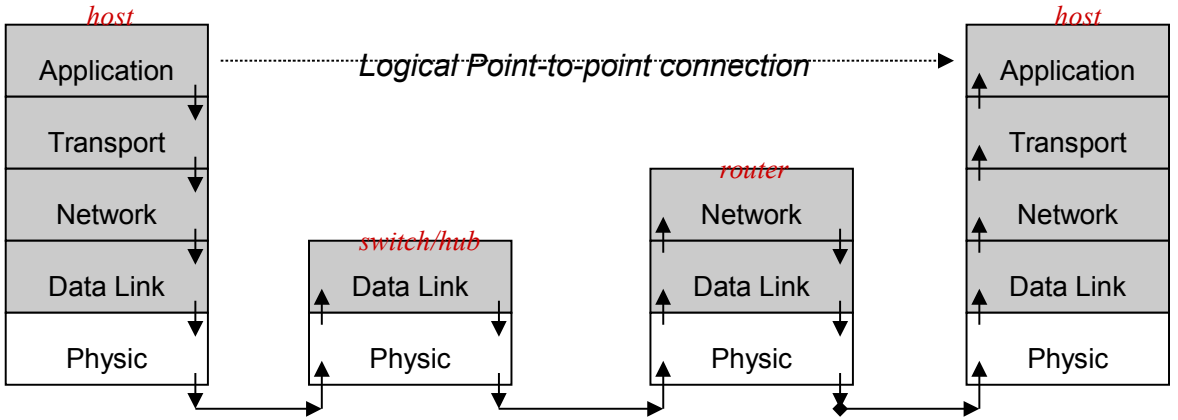
- Transparency
  - Communication inter/intra machines is the same
- Compatibility
  - With existing comunciation mechanims

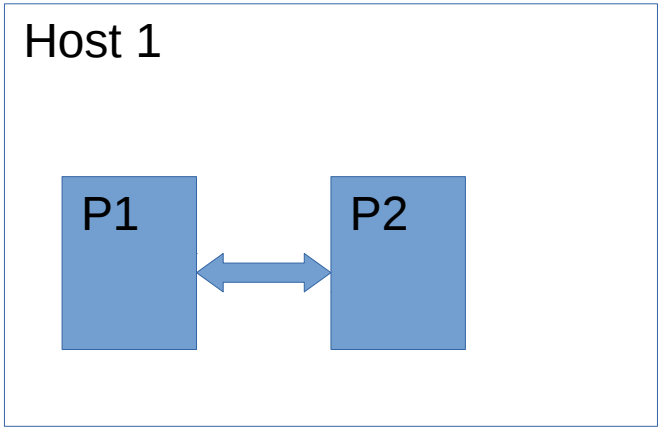
- Stream oriented
- Message Oriented
- Use of File system mechanisms

SOCKETS

- Transparency







SOCKETS

- Compatibility
  - Use of files
    - Used to reference communication channels
  - Use of the Regular I/O API
    - Send/receive data

SOCKETS

- Message oriented
  - Each write is a message
  - No message interleaving in the channel
  - Each message is read atomically
- No data interleaving
  - Concurrent writes to not affect each other
- Atomic reads
  - A read is concluded only after the conclusion of the write
- Extends/replaces
  - mailboxes
  - FIFOS

Sockets

- Introduced in 1981 on BSD 4.1
- It is an API
  - that define access points to applications
    - following the client-server architecture
- Sockets programming
  - more complex than files
  - More parameters
  - More system calls
- Main difference between FS base and socket based communication
  - How channels are opened and created.

Definition of a communication point	socket	Telephone
Assignment of a address to a communication point	Bind /address	Assignment of phone number
Listen to incoming connections	listen	connection of a phone to the network
Start connection	connect()	phone call initiator dials destination number
Receiver established connection	accept()	receiver accepts call lifting the handset
Send / receive of data	send(),recv(	talk
End of communication	close()	lowering of handset

Definition of a communication point	socket	Telephone
Assignment of a address to a communication point	Bind /address	Assignment of phone number
Send message	sendt()	Send SMS
Receive message	recvfrom(	Receive SMS
End of communication	close()	Turn off phone

Message Reception

- On other IPC how receives messages?
  - Any process that open the channel
- Does the sender know the identity?
  - No
- How to solve
  - Assign each channel an address
  - Only one process can read “from” one address

Socket Domains

- The same API allows the creation of different sockets
  - AF_UNIX
    - Communication between processes in the same machine
  - AF_INET
    - Communication between processes in different machines
    - IPv4
  - AF_INET6
    - Communication between processes in different machines
    - IPv6

Socket Domains

- Determines the nature of the communication (local/LAN/WAN)
- Determine the format of the addresses
  - AF_UNIX – a string
  - AF_INT – 4 bytes

Socket Types

- The socket type determines
  - The characteristics of communication
    - Delivery guarantees, ordering guarantees, communication directions
- Are defined as constants staring with the SOCK_ prefix
  - **SOCK_STREAM**
    - stream socket / connection oriented
  - **SOCK_DGRAM**
    - datagram socket / connectionless
  - **SOCK_RAW** raw socket
  - **SOCK_SEQPACKET** sequenced packet socket

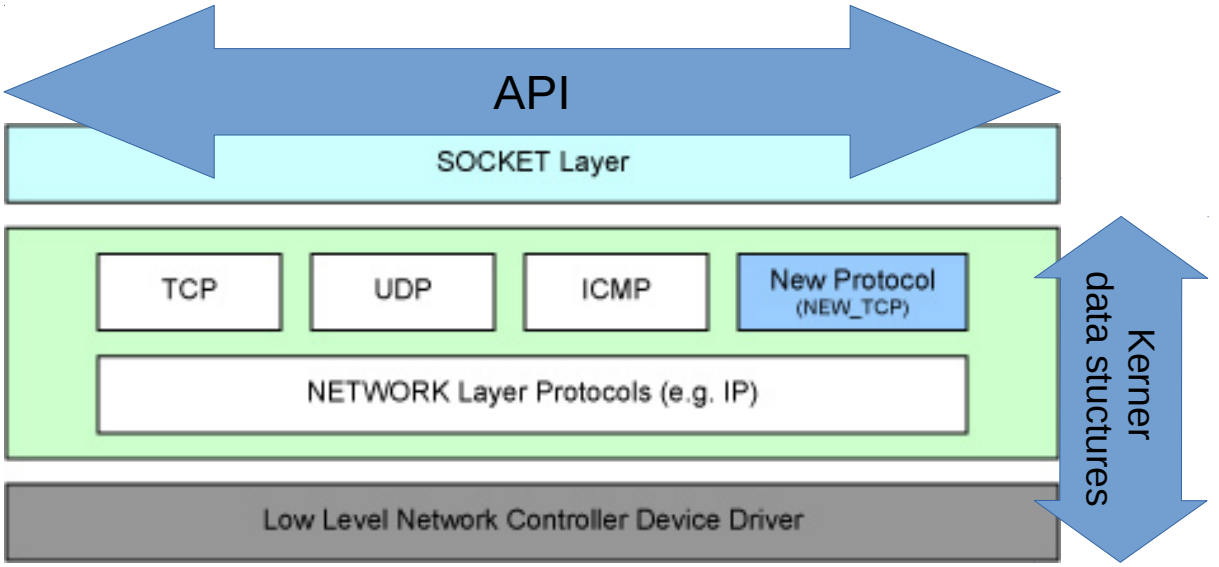
Protocol

- The protocol at transport level depends on the socket Type and domain
  - Not all combinations are possible

		Domain		
		AF_UNIX	AF_INET	AF_UNSPEC
Type	SOCK_STREAM	**YES**	**TCP**	SPP
	SOCK_DGRAM	**YES**	**UDP**	IDP
	SOCK_RAW		**IP**	Sim
	SOCK_SEQPACKET	YES		SPP

Protocol

- Defines
  - Addressing
  - Delivery guarantees
  - Message Structuring
- Affects
  - Kernel data structures
  - API

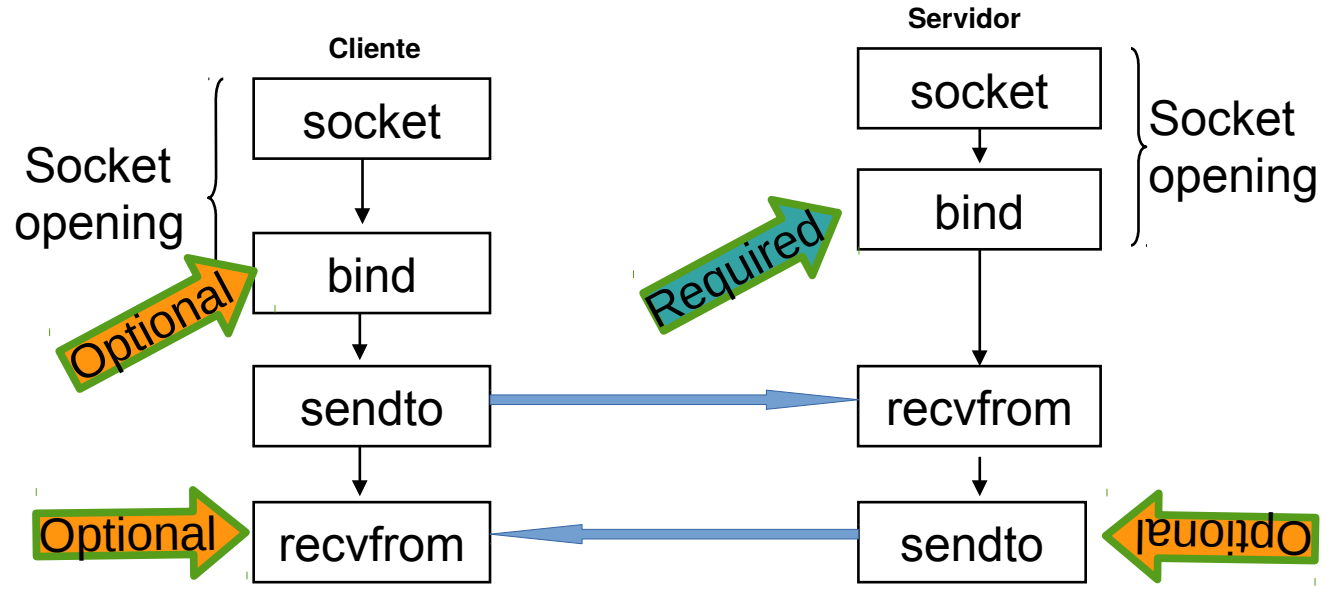


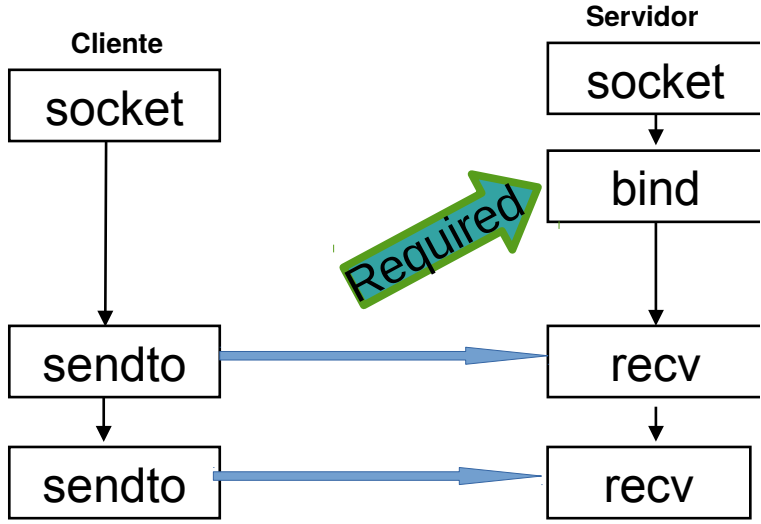
Common socket types

- SOCK_STREAM
  - Reliable delivery
  - Ordered delivery
    - 1st packet to be sent is the 1st to be received
  - Connection oriented
    - Connection setup required before sending messages
  - Bidirectional by default
- SOCK_DGRAM
  - Non reliable delivery
    - Packets can be lost or changed
  - No order guarantee
  - Non existing connection
    - Application should define recipient address for each message
  - (Uni/Bi)directional
    - Recipient can retrieve sender address and reply

Connectionless/datagram sockets

- Client
  - Socket creation
  - Address assignment
  - Message sending
    - Explicit address
- Server
  - Socket creation
  - Address assignment
  - Reception of message

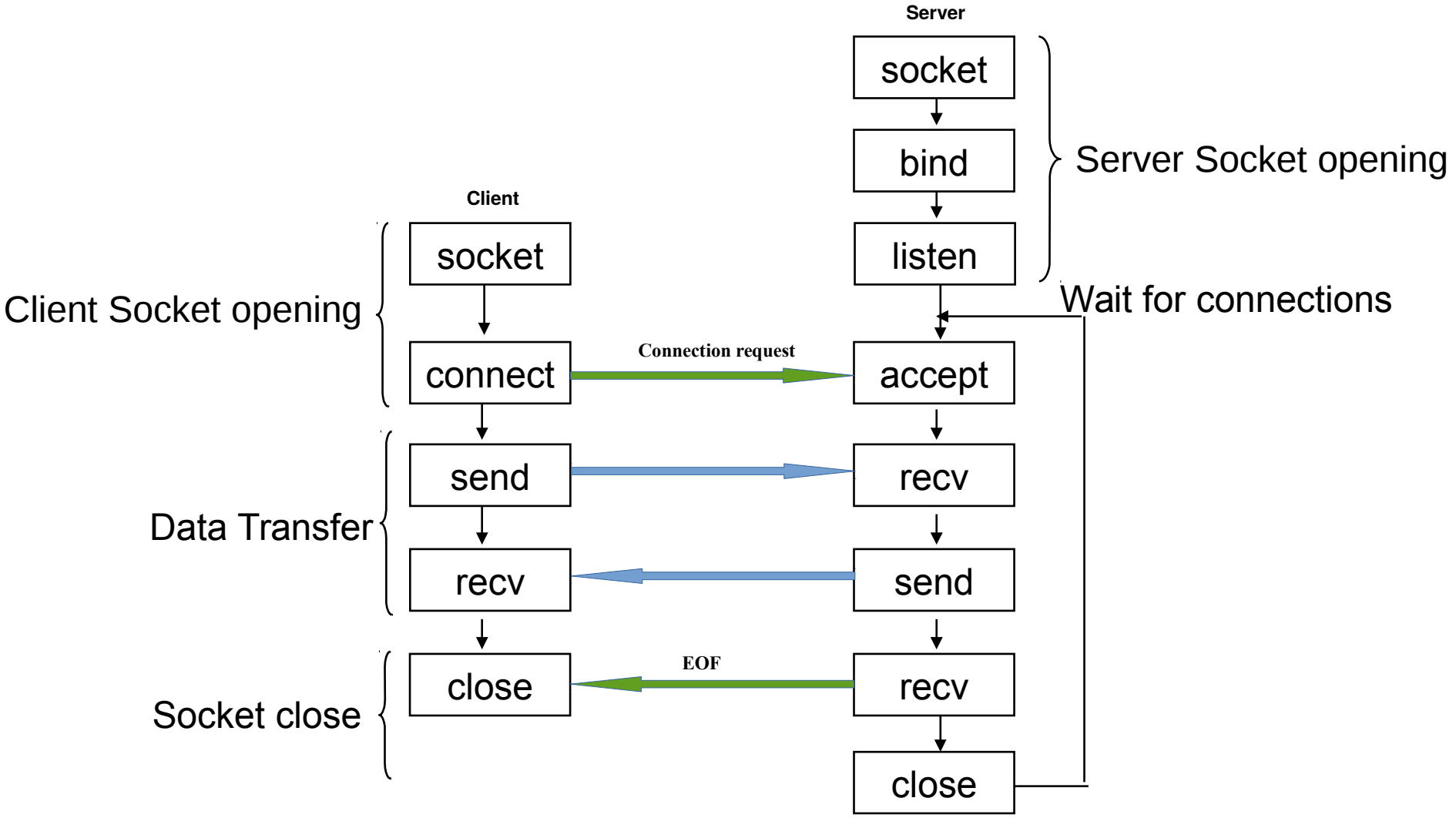




Connection/stream sockets

- Client
  - Socket creation
  - Connection to server
    - Explicit address
  - Message sending
  - Close
- Server
  - Socket creation
  - Address assignment
  - Reception of connections
  - Connection acceptance
  - Reception of message
  - Close

Connection/stream sockets



UNIX Domain Sockets

- Implementation – Kernel / syscall
- Scope - Local
- Duplex
- Time-Coupling
- Space-coupling (Strong!)
- Explicit
- Synchronization – Yes (reads) no (writes)
- Process relation - unrelated
- Identification – file path
- API – specific API + file operation

Sockets address

- Some operations require an address
  - Bind / connect
  - Sendto / recvfrom
- struct sockaddr *src_addr
  -
  - Placeholder for various address classes
    - sockaddr_un - unix
    - sockaddr_in - IP
    - sockaddr_nl - netlink
    - sockaddr_atalk - appletalk

```
struct sockaddr {
        sa_family_t sa_family;
        char        sa_data[14];
}
```

UNIX Domain addresses

- Unix domain addresses are defined using
  - **sockaddr_un** data type
- Definition of the domain
  - sun_family = AF_UNIX
- Definition of the socket path
  - Strcpy(addr.sun_path, “/tmp/sock_1”)

Struct sockaddr_un

sun_family	Family
sun_path	Pathname (up to 108 bytes)

- `#include <sys/un.h>`
- `struct sockaddd_un addr;`
- `addr.sun_family = AF_UNI;`
- `strcpy(addr.sun_path, “/tmp/sock_1”);`

UNIX Domain - Socket creation

- int socket(int domain, int type, int protocol);
  - 1st argument (Domain):
    - AF_UNIX
  - 2nd Argument (type):
    - SOCK_DGRAM / SOCK_STREAM
  - 3rd argument (Identifies the transport protocol):
    - 0 – default value
  - Return the socket descriptor or -1 (in case of error)
    - `if ((s = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1) {`
    - `  perror("socket");`
    - `  exit(1);`
    - `}`

UNIX Domain - Socket creation

- Socket system call
  - Does not determine
    - where the data comes from
    - where the data goes to
- Just creates the communication interface
  - Used to access the channel
- Anonymous
  - Cannot act as server (receive connections/ receive data)
    - Can only connect
    - Can only write/send
- With address
  - Can receive connections (SOCK_STREAM)
  - Can receive messages (SOCK_DGRAM)
  - Address assignment using the bind function

UNIX Domain - Bind

- An address should be assigned to a socket
  - To receive connections
  - To receive messages
- Bind system call
  - int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
  - 1st argument
    - Socket descriptor
  - 2nd parameter
    - Pointer to structure containing the address
    - The structure with the address depends on the protocol
      - sockaddr_in
      - sockaddr_un
  - 3rd argument size of the structure containing the address

Bind

- Returns
  - 0 success
  - -1 error
- `err = bind(s,(struct sockaddr *)&local,`
- `sizeof(local));`
- `if(err == -1) {`
- `perror("bind");`
- `exit(1);`
- `}`

Sendto

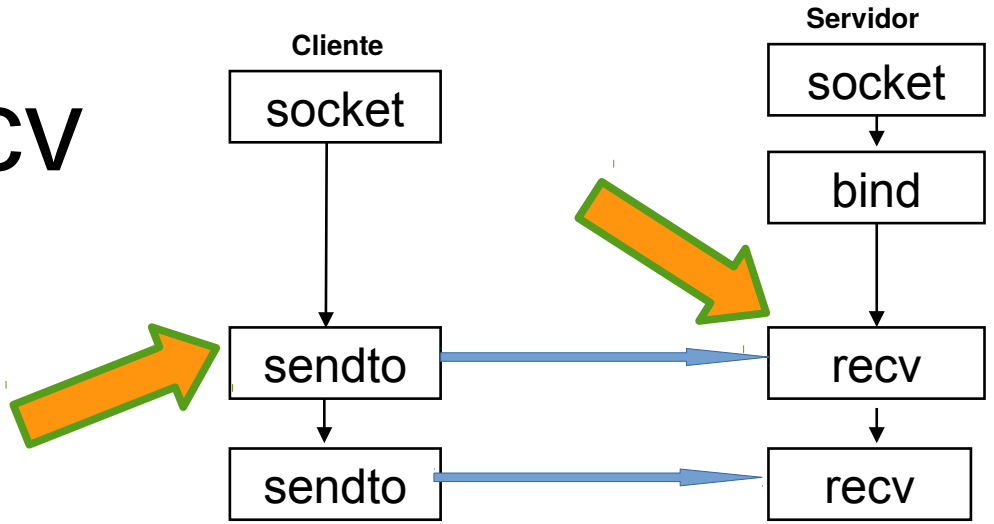
```
ssize_t sendto(int sockfd,
               const void *buf, size_t len,
               int flags,
               const struct sockaddr *dest_addr,
               socklen_t addrlen);
```

- Every time a message is sent the address should be included
  - 1st, 2nd, 3rd arguments
    - Similar to write
  - 4th argument flags (use 0)
  - 5th and 6th Arguments
    - The address of the destination
    - pointer to structure and size of structure

sendto

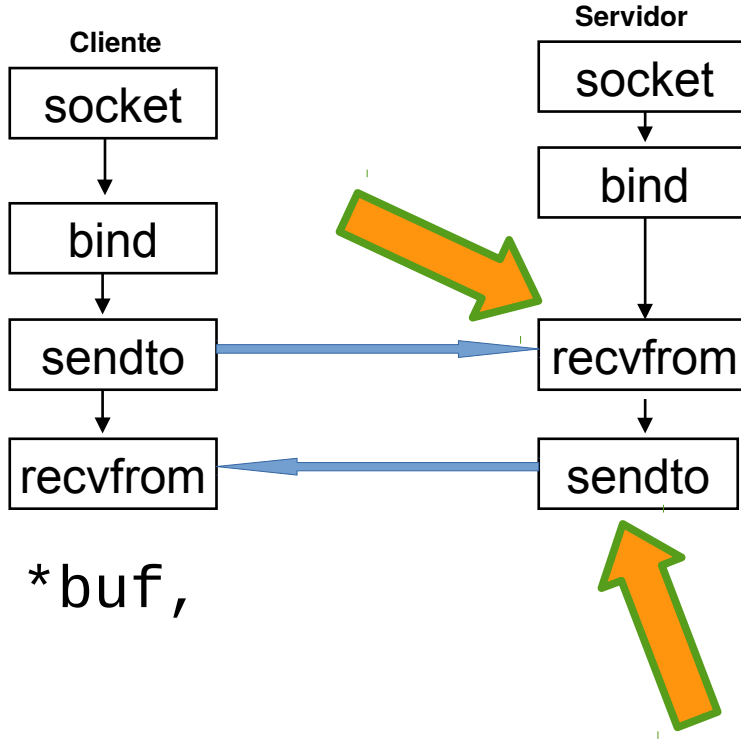
- sendto system call
  - Returns
    - Number of characters sent
    - -1 error (use errno and perror)
- sendto is only used in connectionless sockets
  - **Can not be replaced by write**

recv



- Reception of messages
  - Similar to pipes
  - Read can be used
- ssize_t recv(int sockfd, void *buf, size_t len, int flags);
  - 1st , 2nd and 3rd argument
    - Similar to read
  - 4º flags
    - Usualy 0
  - Retorns the number of received
    - Retunr -1 in case of error (errno).
  - Blocks
  - Reads the first **message** on the socket

recvfrom



- The **recv** can be used but
  - Does not identifies sender
- If the sender address needs to be known:
  - Use function recvfrom
    - After the bind on the client
- `ssize_t recvfrom(int sockfd, void *buf,`
- `size_t len,`
- `int flags,`
- **`struct sockaddr *src_addr,`**
- **`socklen_t *addrlen);`**
  - 4 first arguments like recv
  - 5th argument will store the sender address
  - 6th parameter will store the size of the address

recvfrom

- struct sockaddr_un client_addr;
- socklen_t size_addr;
-
- nbytes = recvfrom(sock_fd, buff, 100, 0,
- **(struct sockaddr *) & client_addr**,
- **&size_addr**);

Closing

- Server and clients should close the sockets
  - In order to orderly end communication
  - #include <sys/socket.h>
  - int close(int);
- The parameter identifies the socket descriptor
- The close system call
  - Closes connections (if using SOCK_STREAM) .
  - Releases used port (if usinf AF_INET)
- **Does not remove the file (if using AF_UNIX)**
  - **Use unlink**

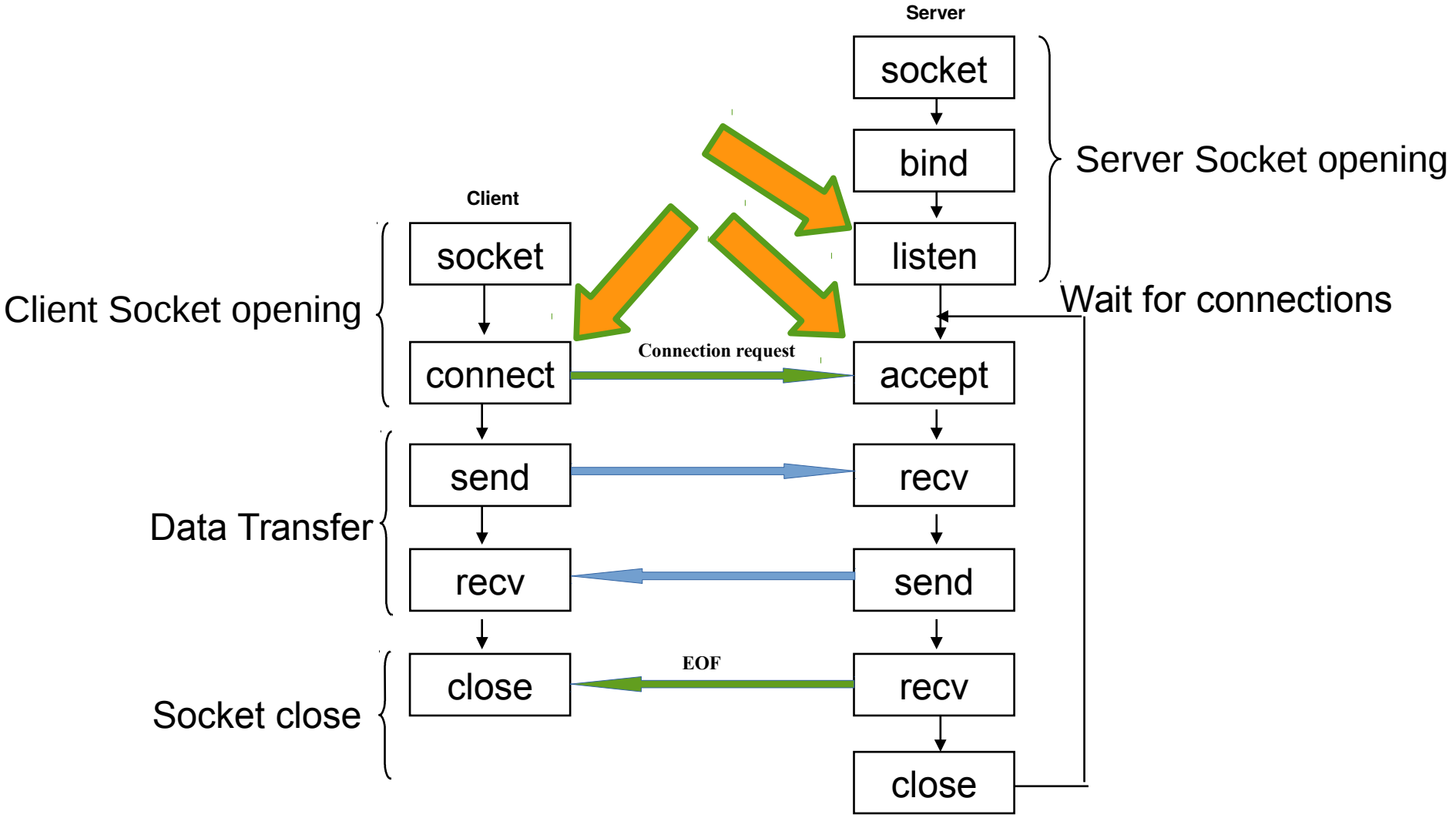
Socket pair

- Parent/child can use UNIX sockets
  - Without assigning an address
  - FD Inherited
  - To replace PIPEs
    - Bidirectional
- int socketpair(int domain, int type, int protocol,
- int sv[2]);
  - Only UNIX Domain

Socket pair

- Int sv[2]
- socketpair(AF_UNIX, type, 0, &sv);
  - SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET
- Process 1
  - Read and write on sv[0]
- Process 2
  - Read and write on sv[1]
- Bidirectional communication
  -

Connection/stream sockets



Domínio UNIX / STREAM

- Message oriented
- With connection
  - Client should connect to the server
  - Does not need to address each message
- With guarantees
  - Delivery
  - Order
  - integrity

- Connection establishing
  - Listen (server)
  - Connect (Client)
  - Accept (Serve
- On the server
  - A new socket is created
    - Dedicated to communication with the client
  - Original socket can receive more connections
- Message transition
  - read/recv
  - Write/send
  - Sendto/recvfrom
    - Not needed

Listen (server)

- The server should inform the operating system that is ready to receive connections
- Sates the willingness to accept incoming connections
- `int listen(int sockfd, int backlog);`
  - 1st arguments identifies the socket.
    - This socket should have been blinded
  - 2nd argument defines the number of client on wait list
    - Between connect and accept
- The listen does not block
- Connection requests received when the list is full
  - Are rejected

Connect (client)

- When using connected sockets
  - SOCK_STREAM ou SOCK_SEQPACKET)
  - The client should establish connection with the server
- The client request connection with the server:
- `int connect(int sockfd,`
- `const struct sockaddr *addr,`
- `socklen_t addrlen);`
  - 1st argument
    - Client socket descriptor
  - 2nd argument
    - server address.
  - 3rd argument
    - size of address

connect

- Connection fails if:
  - **EADDRINUSE**
    - Local address is already in use.
  - **EAFNOSUPPORT**
    - The passed address didn't have the correct address family in its sa_family field.
  - **EBADF**
    - The file descriptor is not a valid index in the descriptor table.
  - **ECONNREFUSED**
    - No-one listening on the remote address.
  - **EFAULT**
    - The socket structure address is outside the user's address space.
  -

- **Connection fails if**
  - **EINTR**
    - The system call was interrupted by a signal that was caught; see signal(7).
  - **EISCONN**
    - The socket is already connected.
  - **ENETUNREACH**
    - Network is unreachable.
  - **ENOTSOCK**
    - The file descriptor is not associated with a socket.
  - **ETIMEDOUT**
    - Timeout while attempting connection. The server may be too busy to accept new connections.

- **man connect**

connect

- The connect call returns
  - In case of error (return -1)
  - In case of success
    - When the server does an accept

Accept

- The server should explicitly accept a connection
- int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
  - 1st parameter
    - Original server socket descriptor (socket, bind, listen)
  - 2nd parameter
    - Stores the client address
  - 3rd parameter
    - Size of the client address
- The **sockfd** is a socket that
  - has been created with socket(2),
  - bound to a local address with bind(2), and is
  - listening for connections after a listen(2).

Accept

- Accept blocks
  - Until a client requests connection
- If 2nd and 3rd arguments are NULL
  - Communication can be done
  - Impossible to know the identity of the client

- The function returns a new socket descriptor
  - That can be use to read/write data to the connected client
- The original socket becomes available to receive new connections
  - Program should do a new accept

Reads/writes

- Using
  - The client sockets
  - The socket created by accept
- There is no need to explicitly address the receiver
  - Already done in the connect/accept
  - read/write
  - send/recv

UNIX Domain / STREAM

- Server
  - s = **socket(AF_UNIX, SOCK_STREAM, 0)**
  - **bind**(s, (struct sockaddr *)&local,
  - sizeof(local));
  - **listen**(s, 10)
  - new_s = **accept**(s, NULL, NULL);
  - n = **recv**(new_s, str, 100, 0);
  - **send**(new_s, str, n, 0);
- Client
  - **s = socket(AF_UNIX, SOCK_STREAM, 0)**
  - **connect**(s, (struct sockaddr *) &server, sizeof(server))
  - **send**(s, str, n, 0);
  - n = **recv**(s, str, 100, 0);

Domínio INET

- Sockets similar to UNIX
  - Same characteristics
  - Same programming work-flow
  - Datagram
    - UDP/IP
  - STREAM
    - TCP
  - RAW
    - IP
  - Different addressing
    - IP Address + Port

IP (TCP/UDP) Sockets

- Same API
  - Socket / bind
  - Listen / connect / accept
  - send(to) recv(from)
- Different address

```
struct sockaddr_in {
    short int sin_family;  // AF_INET
    unsigned short int sin_port; // Port
    struct in_addr sin_addr; // address
    unsigned char sin_zero[8];
};
```

```
struct in_addr {
    uint32_t s_addr;
};
```

IP Addressing

- **#include <netinet/in.h>**
- Sockaddr_in
  - **sin_family :Familia AF_INET**
  - sin_port : porto do serviço
  - sin_addr: endereço

```
struct sockaddr_in {
    short          sin_family;
    u_short        sin_port;      /* número de porto */
    struct in_addr sin_addr;      /* endereço IP */
    char           sin_zero[8];   /* não usado */
};
```

- struct in_addr

```
Struct in_addr{
   u_long s_addr; /* 32 bits network order */
}
```

IP Addresses Enconding

```
int inet_aton(const char *cp, struct in_addr *inp)
```

- String to binary
  - Converts endress (“xxx.yyy.www.zzz”) to binary
  - Stores binary version (32bits) on address structure
  - Returns 0 on error

- char *inet_ntoa(struct in_addr in);
- Converts binary to string
  - After accept/receivfrom ...

IP Addresses - in_addr

```
int main(int argc, char *argv[]) {
   struct in_addr addr;
   if (argc != 2) {
      fprintf(stderr, "%s <dotted-address>\n", argv[0]);
      exit(EXIT_FAILURE);
   }
   if (inet_aton(argv[1], &addr) == 0) {
      fprintf(stderr, "Invalid address\n");
      exit(EXIT_FAILURE);
   }
   printf("%s\n", inet_ntoa(addr));
   exit(EXIT_SUCCESS);
}
```

Server

```
sockaddr_in enderco;


endereco.sin_family = AF_INET;
endereco.sin_port = htons( 22);
```

- Port 22

```
endereco.sin_addr.s_addr = INADDR_ANY;
```

- Any endress

```
bind(fd,(const struct sockaddr *)
    &endereço, sizeof(endereço));
```

Client

```
sockaddr_in enderco;


endereco.sin_family = AF_INET;
endereco.sin_port = htons( 22);
```

- Porto 22

```
inet_aton("146.193.41.1", &endereco.sin_addr)
```

- Endereço do servidor

```
connect(fd,
        (const struct sockaddr *) &endereço,
        sizeof(endereço))
```

IP Addresses

- In the AF_INET domain, the constant INADDR_ANY
  - Determines that the socket is associated to all addresses in the local node.
  - Example:
    - A firewall has different network adapters, on connected to the interent other to the local network
  - **`endereco.sin_addr.s_addr = INADDR_ANY`**

byteorder

- Byte oder on >16bits numbers depend on the processor architecture
- Can be done in two ways:
  - Big-endian:
    - lower addresses with higher order bits (ex: ARM *).
  - Little-endian:
    - lower addresses with lower order bits (ex: Intel x86).
- Integer 1000465 (0x000F4411),

Big-endian			Little-endian	
0x10003	11		0x10003	00
0x10002	44		0x10002	0F
0x10001	0F		0x10001	44
0x10000	00		0x10000	11

htons htonl
ntohs ntohl

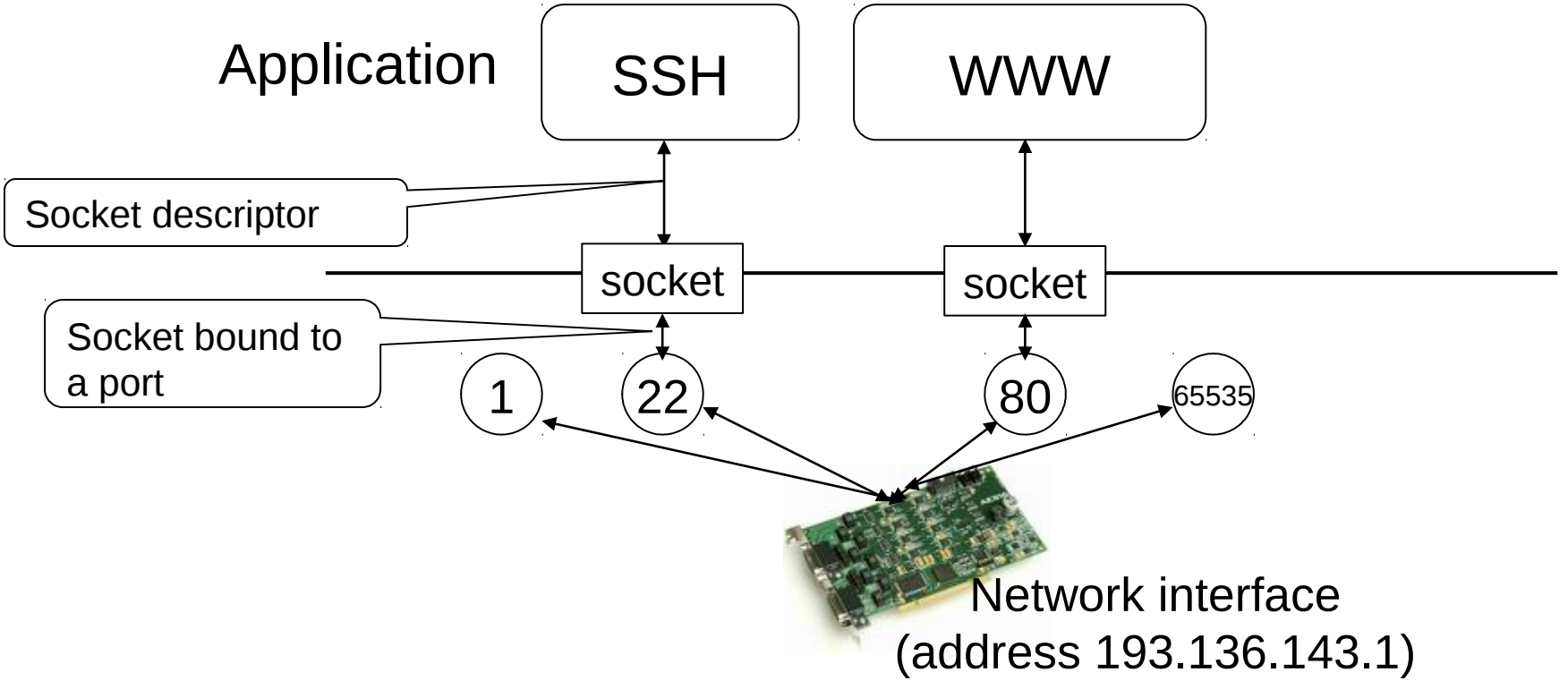
- Big-endian advantages:
  - Integers are stored in the same order as strings (from left to right).
  - Number signal is on the “first byte” (base address) .
- Little-endian advantages:
  - Eases conversion between different length integers (ex: 12 is represented by 0x0C eor 0x000C).
- In the Internet,
  - Addresses are alwayes big-endian.
- The first ARPANET routers (named Interface Message Processor) were 16 bits Honeywell DDP-516 computers big-endian representation

htons htonl
ntohs ntohl

- uint32_t htonl(uint32_t hostlong);
  - The htonl() function converts the unsigned integer hostlong from host byte order tonetwork byte order.
- uint16_t htons(uint16_t hostshort);
  - The htons() function converts the unsigned short integer hostshort from host byte order to network byte order.
- uint32_t ntohl(uint32_t netlong);
  - The ntohl() function converts the unsigned integer netlong from network byte order to host byte order.
- uint16_t ntohs(uint16_t netshort);
  - The ntohs() function converts the unsigned short integer netshort from network byte order to host byte order.
- On the i386 the host byte order is Least Significant Byte first, whereas the network byte order, as used on the Internet, is Most Significant Byte first.

Portos

- A service is dentified by
  - Network address + Port
- The transmition/receptio of mdata is made using a port.
- A socket can be connected on of 64K ports.
- The first 1K portos (1-1023) are reserved by IANA to specific services (listaded in /etc/services) and require root previleges:
  - 22 : SSH
  - 53 : DNS
  - 80 : WWW
  - 115 : secure FTP
  - 443 : secure WWW
- http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers
- Ports in [49152-65535] should not be used by servers
  - Are assigned dynamically to client sockets (accept)


Application
SSH
WWW
Socket descriptor
socket
socket
Socket bound to a port
1
22
80
65535
Network interface
(address 193.136.143.1)

gethostbyname

- Translates names to addresses
  - gasolina.gsd.inesc-id.pt → 146.193.41.15
  - `struct hostent *gethostbyname(const char *name);`
- The name argument can be a "dot-notation" adress or name.
- Returns:
  - `typedef struct {`
  - `  char *h_name;  /* Official name of host.`
  - `  char **h_aliases;  /* Alias list.`
  - `  int h_addrtype;  /* Host address type.`
  - `  int h_length;  /* Length of address.`
  - `  char **h_addr_list;/* List of addresses from name server`
  - `} hostent ;`
  - `#define h_addr h_addr_list[0] /* Address,for backward compat`

gethostbyname

```
struct hostent * hostinfo;
Strcut sockaddr_in address;
address.sin_family = AF_INET
address.sin_port = htons(8088);
hostinfo =
        gethostbyname(“www.gsd.inesc-id.pt”);
address.sin_addr =
        *(struct in_addr *) hostinfo->h_addr;
```

- Man host
- Man endian
- Man INET
- man host
- `man gethostbyname`
- `Man 7 ip`
- `Man 7 tcp`
- `Man 7 udp`

Common Errors

- Common errors with socket programming:
  - Incorrect byte ordering
    - Not calling hton() e ntoh().
  - Disagree on data size and limits (fix or variable).
  - Non initialization of len (recvfrom, accept)
  - Locks
    - Application level protocol not well defined

Blocking calls

- Most socket API calls are blocking
  - When the processe calls such function it gets blocked waiting for an event.
  - accept:
    - Waits for a connection to be received
  - Connect:
    - Waits for the server to accept the connections
  - recv,recvfrom:
    - Waits for data tpo be received
  - send,sendto:
    - Waits for data to be trsnmited to a lower layer.
- In simple application blocking is good:
  - Adds synchronization
  - Limits resource usage (avoids active wait)

Blocking calls

- In complex applications blockingis a problem:
  - Multiple connections are impossible
  - Simultaneous send/recives are difficult
- There are several solutions
  - Multiprogramming
    - Several processes or several threads
      - More complex programming
      - Synchronization required
  - Turn off blocking
    - More complex programming
      - Active wait
  - Use select
    - Wait on multiple descriptors
      - Serializes communication

Blocking calls

```
int select(int,fd_set *,fd_set *,fd_set *, struct
timeval *);
```

- 1st parameter
  - Identifies the number of the highets descriptor + 1.
- 2nd parameter
  - array of reading descriptors (if set select verifies if such descriptors have information to be read)
- 3rd parameter
  - array of writing descriptors (if set select verifies if such descriptors can be writen to).
- 4th parameter
  - array of “exceptions” (if set select verifies if such descriptors has exception.
- 5th parameter defines the waiting interval (NULL to infinite wait).
- The function returns the number of affected descriptors Return -1 in case of error

Blocking calls

- descritors are refered in a bit array of type **struct fd_set** .
- Auxiliary funtions:
  - void FD_ZERO(fd_set *); /* crealr array */
  - void FD_CLR(int, fd_set *); /* set bit to 0 */
  - void FD_SET(int, fd_set *); /* set bit to 1 */
  - int FD_ISSET(int, fd_set *); /* return bit value */
- The select function activates the correct bits on the correct arrays, depending
  - on the input array
  - on the state of the descriptor

Multiple clients

- Select
- Fork
  - FD are Inherited
  - Read retrieve messages from same socket
  - Accept can be done in multiple processes
- Threads