Network programming: sockets

Antonio Lioy < lioy@polito.it >

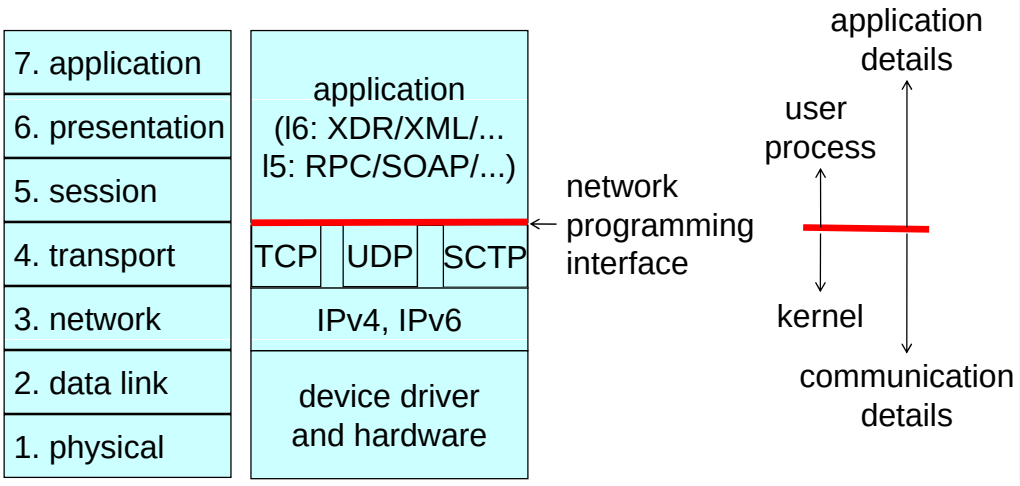
english version created and modified by
Marco D. Aime < m.aime@polito.it >

Politecnico di Torino
Dip. Automatica e Informatica

Warning for programmers

- **network programming is dangerously close to O.S. kernel, and therefore:**
  - It can easily hang the O.S.
    - verify the results of every operation, without assuming anything as granted
  - APIs can vary in details that are minimal but important
    - consider every possible situation to create "portable" programs
    - we will try to use Posix 1.g

ISO/OSI, TCP/IP, network programming



ref. UNP Intro

Exercise – copying data

- **copy the content of file F1 (first parameter on the command line) into file F2 (second parameter on the command line)**

`copyfile.c`

Error messages

- **must contain at least:**
  - [PROG] program name
  - [LEVEL] error level (info, warning, error, bug)
  - [TEXT] error signalling, the most specific as possible (e.g. input file name and line where the problem has occurred)
  - [ERRNO] system error number and/or name (if applicable)
- **suggested format:**

```
( PROG ) LEVEL - TEXT : ERRNO
```

Error functions

- **best to define standard error reporting functions which accept:**
  - a format string for the error
  - a list of parameters to be printed
- **UNP, appendix D.4 (D.3 in 3rd edition)**

	errno?	termination?	log level
`err_msg`	no	no	`LOG_INFO`
`err_quit`	no	`exit(1)`	`LOG_ERR`
`err_ret`	yes	no	`LOG_INFO`
`err_sys`	yes	`exit(1)`	`LOG_ERR`
`err_dump`	yes	`abort( )`	`LOG_ERR`

`errlib.h`
`errlib.c`

stdarg.h

- variable list of arguments (ANSI C)
- declared with an ellipsis (. . .) as the last argument of a function
- arguments used altogether (ap) with special functions (e.g. vprintf, vfprintf, vsprintf, vasprintf, vsnprintf), or separately (va_arg); but in the second case you must know their number by other means

```
#include <stdarg.h>

void va_start (va_list ap, RIGHTMOST);
TYPE va_arg (va_list ap, TYPE);
void va_end (va_list ap);
```

stdarg.h usage example

- create a function named my_printf
- … behaving like printf (thus accepting a variable number of parameters)
- … but printing the string “(MY_PRINTF)” before its output

va_test.c

From 32 to 64 bit: problems

- **migration from 32-bit to 64-bit architecture has changed data sizes**
- **in particular, do not assume anymore that | int | = | pointer |**
- **therefore, pay attention to correctly use the predefined types to avoid problems (e.g. `size_t`)**

	ILP32	LP64
`char`	8	8
`short`	16	16
`int`	32	32
`long`	32	64
pointer	32	64

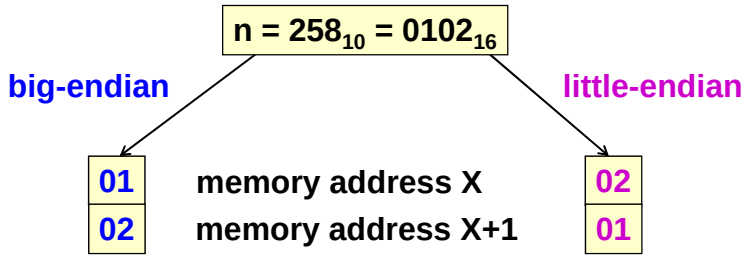
`datasize.c`

Exchanging data between heterogeneous nodes

- **problem:**
  - when exchanging complex data (i.e., not single ASCII characters), do not assume they are encoded in the same way on different nodes
  - data encoding depends on HW + OS
- **solution:**
  - use a neutral format (the network format)
  - some functions automatically do the conversion…
  - … but often it is explicit task for the programmer

Sources of data incompatibility

- different floating-point formats (IEEE-754 / non-IEEE)
- alignment of structs on word boundaries
- byte order of integers (little-endian or big-endian)



byteorder.c

"host to network" functions

- to pass parameters to network functions
- not designed for application data passing

```
#include <sys/types.h>
#include <netinet/in.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);
```

Notes on integer types

- **sometimes you may still find the old types:**
  - u_long (= uint32_t)
  - u_short (= uint16_t)
- **in old versions of cygwin, they were defined as:**
  - u_int32_t (= uint32_t)
  - u_int16_t (= uint16_t)

- **suggestion:**
  - write programs with the types uint32_t and uint16_t
  - when necessary, map them with a conditioned #define

Network addresses

- **IPv4 networks directly use 32-bit addresses**
- **... not names (e.g. www.polito.it), that are translated into addresses by DNS**
- **... and neither dotted addresses (e.g. 130.192.11.51)**
- **... but their 32-bit numerical value**

- **example: 130.192.11.51**
  - = $130*2^{24} + 192*2^{16} + 11*2^{8} + 51$
  - = (((130*256) + 192)*256 + 11)*256 + 51
  - = 130<<24 + 192<<16 + 11<<8 + 51
  - = 2,193,623,859

Network address conversion

- **for generality, standard functions require numerical addresses to be expressed in struct format**
- **to convert numerical addresses from/to strings:**
  - [IPv4] inet_ntoa( ) and inet_aton( )
  - [IPv4 / v6] inet_ntop( ) and inet_pton( )
- **other functions (e.g. inet_addr) are deprecated**

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

struct in_addr { in_addr_t s_addr };

struct in6_addr { uint8_t s6_addr[16]; };
```

inet_aton ( )

- **converts an IPv4 address …**
- **… from "dotted notation" string**
- **… to numerical (network) format**
- **returns 0 if the address is invalid**
- **the string may be composed by decimal numbers (default), octal (start with 0) or hexadecimal (start with 0x) thus 226.000.000.037 is equal to 226.0.0.31**

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

int inet_aton (
  const char *strptr,
  struct in_addr *addrptr
);
```

inet_ntoa ( )

- converts an IPv4 address …
- … from numerical (network) format
- … to “dotted notation” string
- returns the pointer to the string or NULL if the address is invalid
- beware! the pointer being returned points to a static memory area internal to the function

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

char *inet_ntoa (
  struct in_addr addr
);
```

Example: address validation

write a program that:

- accepts a dotted notation IPv4 address on the command line
- returns its numerical (network) value
- signals an error in case of erroneous format or illegal address

verify:

- remember that A.B.C.D = A<<24 + B<<16 + C<<8 + D

avrfy.c
avrfy2.c

inet_pton( )

- converts an IPv4 / IPv6 address …
- … from “dotted notation” string
- … to numerical (network) format
- returns 0 if the address is invalid, -1 if the address family is unknown
- address family = AF_INET or AF_INET6

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

int inet_pton (
  int family,
  const char *strptr,
  void *addrptr
);
```

inet_ntop( )

- converts an IPv4 / IPv6 address …
- … from numerical (network) format
- … to “dotted notation” string
- returns the pointer to the string or NULL if the address is invalid

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

char *inet_ntop (
  int family,
  const void *addrptr,
  char *strptr, size_t length
);
```

Example: address validation

write a program that:

- **accepts an IPv4 or IPv6 address on the command line**
- **tells if it is valid in one of the two IP versions**

`avrfy46.c`

Address sizes

- **to size the string representation of v4/v6 addresses, use macros defined in <netinet/in.h>**

```
#include <netinet/in.h>

#define INET_ADDRSTRLEN  16
#define INET6_ADDRSTRLEN 46
```

IPv6 addresses

- 128 bit = 8 16-bit groups, separated by " : "
- complete form = 1080:0:0:0:8:800:200C:417A
- zeros compression = 1080::8:800:200C:417A

Data structure initialization (ANSI)

- to handle data structures as byte sequences
- … which may include NUL, and therefore cannot be handled with "str…" functions (strcpy, strcmp, …)

```
#include <string.h>

void *memset (
  void *dest, int c, size_t nbyte )
void *memcpy (
  void *dest, const void *src, size_t nbyte )
int *memcmp (
  const void *ptr1, const void *ptr2, size_t nbyte )
```

Data structure initialization (BSD)

- **pre-ANSI functions, defined in Unix BSD**
- **still frequently used**

```
#include <strings.h>

void bzero (
  void *dest, size_t nbyte )
void bcopy (
  const void *src, void *dest, size_t nbyte )
int bcmp (
  const void *ptr1, const void *ptr2, size_t nbyte )
```

Signal management (*)

- **signals are asynchronous events**
- **every received signal corresponds to an implicit default behaviour**
  - often the receiving process is terminated
  - few signals are ignored by default
- **to change the response to a signal (i.e. its disposition):**
  - reset its default behaviour
  - ignore it
  - catch (i.e. intercept) the signal and register a signal handler

Timeout

- **sometimes timeouts are needed to:**
  - wait in idle state for a fixed time (sleep)
  - know when a certain time elapsed (alarm)

`sveglia.c`

sleep ( )

- **starts a timer and suspends the process for the selected time**
- **if terminates because the selected time has expired**
  - returns zero
- **if terminates for being interrupted by a signal**
  - returns the time missing to the requested term

```
#include <unistd.h>
unsigned int sleep (
  unsigned int seconds
);
```

alarm ( )

- starts a timer which generates the SIGALRM signal at expiration time
- the process is not suspended
- note: the default response to SIGALRM is terminating the receiving process
- a new call replaces the current timer (e.g. use `alarm(0)` to clear the current timer)
- (!) unique timer for all processes in the group

```
#include <unistd.h>
unsigned int alarm (
  unsigned int seconds
);
```

Signal management

- pay attention to semantic differences in various operating systems
- for 'signum' use the signal logical names (SIGCHLD, SIGSTOP, …)
- the handler can be:
  - a user-defined function
  - SIG_IGN (ignore the signal)
  - SIG_DFL (default behaviour)

```
#include <signal.h>
typedef void Sigfunc(int);
Sigfunc *signal (int signum, Sigfunc *handler);
```

signal() function

- **signal() is a pre-POSIX function**
  - its behaviour varies across Unix versions, and also varied historically across different releases
  - the only portable use is setting a signal's disposition to SIG_DFL or SIG_IGN
- **POSIX has defined the new sigaction() function**
  - most implementations of signal() now call sigaction()
- **see UNPv3 chap. 5.8**

Notes on signals

- **SIGKILL and SIGSTOP cannot be intercepted nor ignored**
- **SIGCHLD and SIGURG are ignored by default**
- **in Unix, use `kill -l` to list all signals**
- **in Unix, signals can be generated also manually via**
  `kill -`*signal pid*
- **for example, to send SIGHUP to process 1234:**

```
kill -HUP 1234
```

Standard signals (POSIX.1)

name	*value*	*action*	*notes*
HUP	1	term	hangup of controlling terminal or death of controlling process
INT	2	term	interrupt from keyboard
QUIT	3	core	quit from keyboard
ILL	4	core	illegal instruction
ABRT	6	core	abort signal from abort( )
FPE	8	core	floating-point exception
KILL	9	term	kill
SEGV	11	core	invalid memory reference
PIPE	13	term	broken pipe (write to pipe w/o readers)
ALRM	14	term	timer signal from alarm( )
TERM	15	term	termination signal

Standard signals (POSIX.1) – cont.

name	*value*	*action*	*notes*
USR1	16/10/30	term	user-defined signal 1
USR2	17/12/31	term	user-defined signal 2
CHLD	18/17/20	ignore	child stopped / terminated
CONT	25/18/19	cont	continue if stopped
STOP	23/19/17	stop	stop process
TSTP	24/20/18	stop	stop from tty
TTIN	26/21/21	stop	tty input for background process
TTOU	27/22/22	stop	tty output from background process

kill( )

- **to send signals between processes**
- **returns 0 if OK, -1 in case of error**
- **(pid=0) sends signals to all processes in the group**
  - often used to send signals to all the own children
- **(pid<0) sends signals to all processes in the group "–pid"**
- **(signum=0) does not send any signal but does error control (inexistent process or group)**

```
#include <sys/types.h>
#include <signal.h>
int kill ( pid_t pid, int signum );
```

Socket

- **is the base primitive for TCP/IP communications**
- **is the endpoint of a communication**
- **support channel-oriented communications:**
  - connected sockets (a pair of connected sockets provides a bidirectional interface of *pipe* type)
  - one-to-one model
- **support message-oriented communications:**
  - connectionless sockets
  - many-to-many model

Socket types

- **three fundamental types:**
  - STREAM socket
  - DATAGRAM socket
  - RAW socket
- **typically:**
  - stream and datagram used at application level
  - raw used for protocol development (access to every field of the IP packet, including the header)

Stream socket

- **octet stream**
- **bidirectional**
- **reliable**
- **sequential flow**
- **flow without duplications**
- **messages with unbounded size**
- **sequential file interface**
- **usually used for TCP channels**

Datagram socket

- message-oriented
- message = unstructured set of octects (binary blob)
- bidirectional
- not sequential
- not reliable
- messages possibly duplicated
- messages limited to 8 KB
- dedicated interface (message based)
- usually used for UDP or IP packets

Raw socket

- provides access to the underlying communication protocol
- usually of datagram type
- complex programming interface (and often OS dependent): not designed for distributed application programmers
- used for protocol development

Communication domain

- **every socket is defined within a communication domain**
- **a domain is an abstraction that implies:**
  - an addressing structure: the “Address Family” (AF)
  - a protocol suite implementing the sockets within the domain: the “Protocol Family” (PF)
- **by convention, it is frequent to use AF only (because of bi-univocal correspondence with PF)**

Binding

- **sockets are created without any identifier**
- **no process can refer or access a socket without an identifier**
- **before being used, a socket must be associated with an identifier**
  - = network address (for network sockets)
  - = logical name (for OS sockets)
- **“binding” sets the socket address and makes the socket accessible to the network**
- **binding depends on the protocol being used**
- **usually, only servers do binding explicitly (for clients the address is set by the OS based on routing)**

Association (INET domain)

- to let two processes communicate on a network, an association must exists between them
- in the AF_INET domain an association is a quintuple
- every quintuple must be unique

protocol (TCP, UDP, …)
(local) IP address
(local) port
(remote) IP address
(remote) port

Association (Unix domain)

- two process on the same Unix node can communicate using a local association
- in the AF_UNIX domain, associations are triples
- every triple must be unique

protocol (channel, msg)
pathname (local)
pathname (remote)

Associations

- **created with the system call bind( ) which actually creates one half of the association**
- **the half-association is then completed:**
  - by the server with accept( ), that removes a request form the queue and creates a new socket dedicated to the connection; it's a blocking call
  - by the client with connect( ), that may also assign the local port (and address); it's a blocking call

Connected sockets (stream)

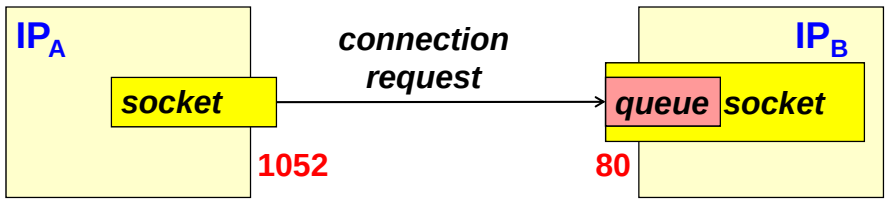
- **creating new connections is typically an asymmetric operation**
- **every process creates its own endpoint with socket()**
- **the server:**
  - assigns an identifier to the socket using bind( )
  - starts listening on the socket by calling listen( )
  - when a connection request arrives, accepts it with accept( ) which removes the request from the queue and creates a new socket dedicated to the connection
- **the client connects to the server with connect( ), that also performs implicit binding by assigning the local port (and address)**

Stream socket: pre-connection

Client:
1. creates a socket

Server:
1. creates a socket
2. associates a port to the socket
3. starts listening on the queue

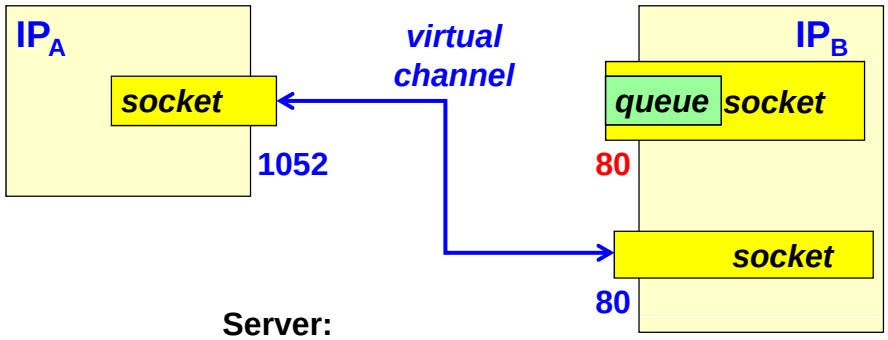
Stream socket: connection request



Client:
1. requests connection to server' s port

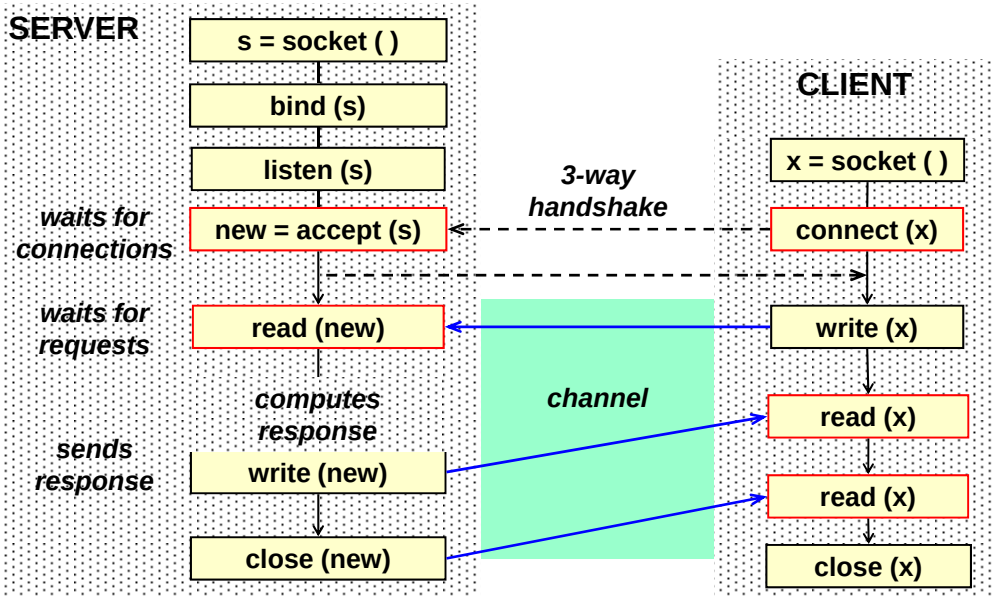
Server:
1. receives connection request

Stream socket: established connection

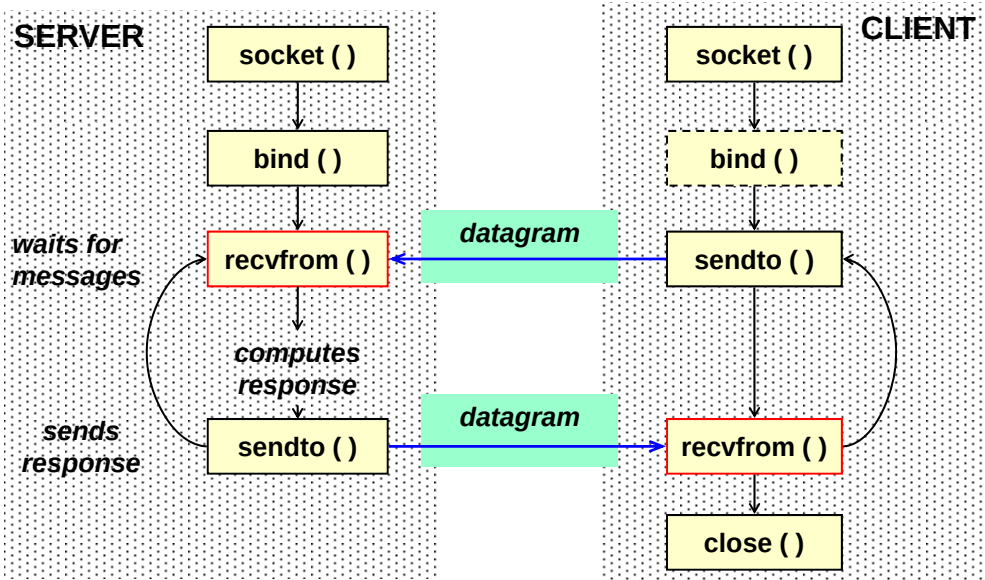


Server:
1. accepts the request
2. completes the channel with a new socket
3. returns waiting on the queue of the original socket

Stream socket: logical flow



Datagram socket: logical flow



Datagram socket - differences

- **allow:**
  - exchanging data without connection (messages include destination and source addresses)
  - sending from one socket to multiple destinations
  - receiving on one socket from multiple destinations
- **therefore, in general, the model is "many-to-many"**
- **the 'client' and 'server' terms are only meaningful for the application level**
- **there are no differences in the calls made by the various processes involved in the communication (symmetric)**

Socket in C

The gcc compiler

- one of the best compilers
- use the flag " – Wall –Werror" to avoid missing potential error sources
- attention: the warning for using uninitialized variables is triggered only when optimising the program (at least with –O1)

Notes for compiling

- **in Solaris > 7, you need to link several libraries:**
  - libsocket
    - base socket functions
  - libnsl (name services library)
    - inet_addr (and more)
  - libresolv (name resolver library)
    - h_errno, hstrerror (and more)

```
gcc -lsocket -lnsl -lresolv ...
```

make and makefile

- **rules in files named “Makefile”**
  - target: dependencies ...
  - TAB command_to_create_the_target
- **$ make [*target*]**
  - the first target is the default one
- **example:**

```
prova.exe: prova.o mylib.o
     gcc -o prova.exe prova.o lib.o
prova.o: prova.c mylib.h
     gcc -o prova.o -c prova.c
mylib.o: mylib.c mylib.h
     gcc -o mylib.o -c mylib.c
```

Dependencies

- **gcc –MM file**
  - analyses file and creates its list of dependencies in the make format, ignoring system headers
  - very useful to automatically create and include dependencies (with the command "make depend")

```
depend:
	rm -f depend.txt
	for f in $(SRC); do gcc -MM $$f>>depend.txt; done

# dependencies
-include depend.txt
```

Application data encoding

- **ASCII encoding:**
  - sscanf / snprintf to read / write ASCII data (only for C strings, i.e. terminated by \0)
  - memcpy (not strings but the length is known)
- **binary encoding:**
  - hton / ntoh / XDR for reading / writing binary data
- **examples (n_{10}=2252):**

```
ASCII (4 bytes)
'2'  '2'  '5'   '2'
0x32 0x32 0x35  0x32
```

```
bin (16 bits)

0x08 0xCC
```

Application data: formats

- **fixed format:**
  - read / write the specified number of bytes
  - convert bytes according to their encoding
- **variable format (with separators and terminator):**
  - read up to the terminator, then convert
  - pay attention to overflow in case terminator is missing
- **variable format (TLV = tag-length-value):**
  - read N bytes according to length
  - convert according to the format specified by tag

Unix socket descriptor

- **standard file descriptor referred to a socket instead of a file**
- **can be used for read or write operations with normal direct I/O functions**
- **it's possible to use the system calls working on files**
  - close, read, write
  - exception: seek
- **other system calls are available for the peculiar functions of sockets (i.e. not applicable to files)**
  - send, recv, …
  - sendto, recvfrom, …

socket( )

- **creates a socket**
- **returns the socket descriptor in case of success, -1 in case of error**
- **family = constant with type AF_x**
- **type = constant with type SOCK_x**
- **protocol = 0 (except in raw sockets)**

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

socket( ): parameters

- **family:**
  - AF_INET
  - AF_INET6
  - AF_LOCAL (AF_UNIX)
  - AF_ROUTE
  - AF_KEY
- **type:**
  - SOCK_STREAM
  - SOCK_DGRAM
  - SOCK_RAW
  - SOCK_PACKET (Linux, access to layer 2)

Possible combinations

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	yes		
SOCK_DGRAM	UDP	UDP	yes		
SOCK_RAW	IPv4	IPv6		yes	yes

socket( ) : wrapper

- useful to write test once and reuse, instead of repeating them every time

```
int Socket (int family, int type, int protocol)
{
  int n;
  if ( (n = socket(family,type,protocol)) < 0)
    err_sys ("(%s) error - socket() failed", prog);
  return n;
}
```

The “wrapper” concept

- **given a function to “wrap”, create a new function:**
  - with the same name but starting with capitalised letter
  - with the same parameters
  - of void type (not when the return value is the response of the function), since controls are made within the new function
- **do not invent “exotic” controls; just read carefully the function description and run the controls on the return value (or on other mechanisms used for error signalling)**

Wrapper example

- **in place of strcpy( ) always prefer strncpy( )**
- **… but also this function can generate errors:**

```
char *strncpy (char *DST, const char *SRC, size_t LENGTH);

DESCRIPTION
'strncpy' copies not more than LENGTH characters from the
string pointed to by SRC (including the terminating null
character) to the array  pointed  to  by DST.

RETURNS
This function returns the initial value of DST.
```

Wrapper for strncpy

```
void Strncpy (
  char *DST, const char *SRC, size_t LENGTH)
{
  char *ret = strncpy(DST,SRC,LENGTH);
  if (ret != DST)
    err_quit(
      "(%s) library bug - strncpy() failed", prog);
}
```

Using “wrapped” functions

- when an error occurs, typically the wrapper reports it and terminates the calling process
- if the caller is a child of a concurrent server, there is usually no problem
- if the caller is the parent process of a concurrent server or it is a client that should continue interacting with the user, then terminating the caller may not be the correct thing to do ...

Socket addresses

- use the sockaddr struct which is the address of a generic socket (Internet, Unix, …)
- actually, it is just an overlay for the specific cases (sockaddr_in, sockaddr_un, …)
- defined in <sys/socket.h>

```
struct sockaddr {
  uint8_t     sa_len;      // not mandatory
  sa_family_t sa_family;   // AF_xxx
  char        sa_data[14]  // identifier
}
```

Internet socket addresses

- level 3 network address (in network format)
- level 4 port (in network format)
- the level 4 protocol is defined implicitly based on the socket type (STREAM, DATAGRAM)

```
struct sockaddr_in
{
  uint8_t         sin_len;      // not mandatory
  sa_family_t     sin_family;   // AF_INET
  in_port_t       sin_port;     // TCP or UDP port
  struct in_addr  sin_addr;     // IP address
  char            sin_zero[8]   // unused
}
```

connect( )

- create a connection between a “local” socket and a “remote” socket, specified through its identifier (=address and port)
- in practice it starts the TCP 3-way handshake
- the OS automatically assigns to the local socket a proper identifier (address and port)
- returns 0 if OK, -1 in case of error

```
#include <sys/socket.h>
int connect (int sockfd,
  const struct sockaddr *srvaddr, socklen_t addrlen)
```

bind( )

- assigns an identifier (address and port) to a socket
- returns 0 in case of success, -1 in case of error
- if the IP address is not specified, the kernel assigns it based on the received SYN packet
- if the port is not specified, the kernl assigns an ephemeral port
- INADDR_ANY to specify any address

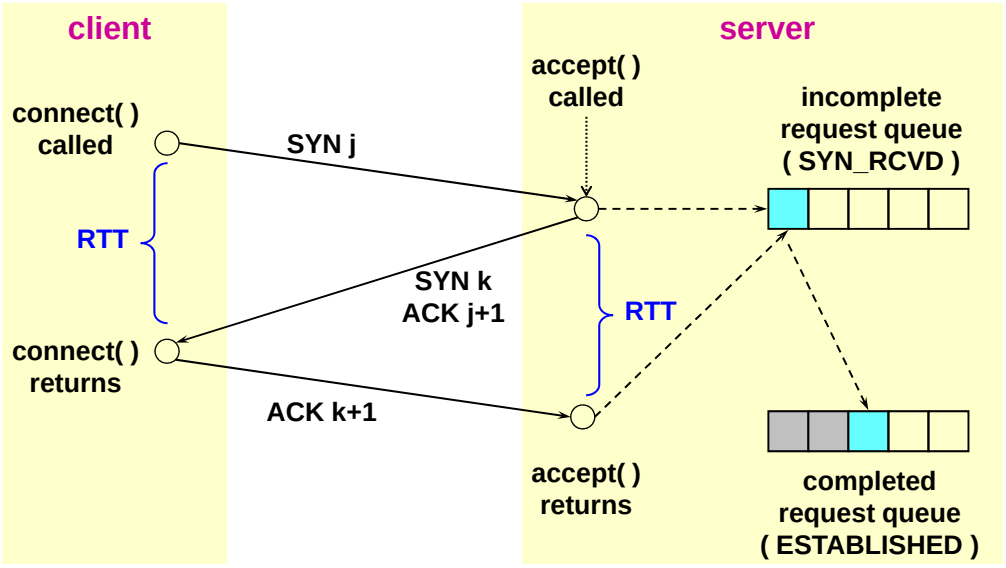
```
#include <sys/socket.h>
int bind ( int sockfd,
  const struct sockaddr *myaddr,
  socklen_t myaddrlen )
```

listen( )

- **transforms a socket from active to passive**
- **specifies the size of the queue for pending requests (= sum of the two queues, plus sometimes a 1.5 adjusting factor)**
- **critical factor for:**
  - high load servers
  - resist to "SYN flooding" attacks
- **returns 0 in case of success, -1 otherwise**

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

Size of which queue?



Notes on “backlog” parameters

- **Linux (starting from 2.2) and Solaris (from 2.9) use the SYN_COOKIE mechanism and thus the parameter specifies only the length of the Established queue**
- **the length of the SYN_RCVD queue:**
  - automatically sized (256-1024 entries) based on available RAM
  - configurable with sysconf or …/ip/…
  - infinite when SYN_COOKIE is active (default from kernel 2.4)

on SYN_COOKIE see https://cr.yp.to/syncookies.html

listen( ) : wrapper

- **useful to avoid fixing the queue dimension in the code but make it configurable (via arguments or environment variables)**
- **for example, with a wrapper:**

```
#include <stdlib.h> // getenv()

void Listen (int sockfd, int backlog)
{
  char *ptr;
  if ( (ptr = getenv("LISTENQ")) != NULL)
    backlog = atoi(ptr);
  if ( listen(sockfd,backlog) < 0 )
    err_sys ("(%s) error - listen failed", prog);
}
```

Reading environment variables in C

- **use the function getenv( )**
- **receives in input the name of the environment variable as a string**
- **returns the pointer to the string associated to the value**
- **... or NULL if the variable is undefined**

```
#include <stdlib.h>
char *getenv (const char *varname);
```

Environment variable listing in C

- **"envp" parameter passed to main( ) function**
  - supported by MS-VC++ and gcc … but not ANSI
- **envp is an array of string pointers**
  - every string contains the pair:
    - VARIABLE_NAME=VARIABILE_VALUE
  - the last element in the array has NULL value
    - necessary, since there is no parameter to report the number of the strings

```
int main (int argc, char *argv[], char *envp[]);
```

Setting environment vars (in BASH)

- **export VARNAME=VARVALUE**
  - adds a variable to the execution environment
- **export –n VARNAME**
  - removes the variable from the execution environment
- **env VARNAME=VARVALUE COMMAND**
  - executes the command temporarily inserting the variable in its environment
- **printenv [VARNAME]**
  - lists all environment variables or the selected one
- **note: “export” is built-in in bash while “env” and “printenv” are external commands (in /usr/bin)**

Examples of reading environment variables

- **printevar.c prints the environment variable value whose name is given on the command line**
- **prallenv.c prints names and values of all defined environment variables**

`printevar.c`
`prallenv.c`

accept( )

- retrieves the first connection available in the completed request queue
- blocks in case there are no pending requests (exception: if the socket is not blocking)
- returns a new socket descriptor, connected with the client's one
- side effect: returns in cliaddr the identifier of the connected client (but when you provide NULL)

```
#include <sys/socket.h>
int accept (int listen_sockfd,
  struct sockaddr *cliaddr, socklen_t *addrlenp)
```

close( )

- immediately closes the socket
- the socket is no more usable by the process, but the kernel will try to send any data already queued to be sent, before closing the TCP channel
- behaviour configurable with SO_LINGER
- returns 0 when terminating with success, -1 in case of error

```
#include <unistd.h>
int close ( int sockfd )
```

Stream communication

- **use unbuffered I/O functions to avoid**
  - waiting indefinitely (input)
  - terminating prematurely in case of NUL (output)
- **never use the standard I/O library <stdio.h>**
- **in particular, use the system calls read( ) and write( ):**
  - work on file descriptors
  - returns the number of bytes that have been read / written, -1 in case of error

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t nbyte )
ssize_t write(int fd, const void *buf, size_t nbyte)
```

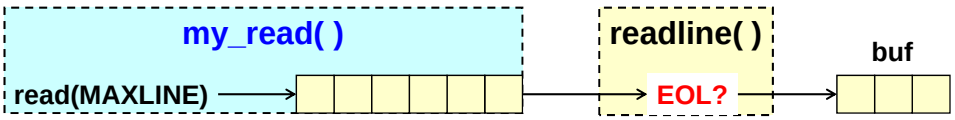
Result of a read on a socket

- **greater then zero: number of received bytes**
- **equal to zero : closed socket (EOF)**
- **lower then zero : error**

- **attention:**
  - due to fragmentation and buffering …
  - … the number of read bytes can be lower than expected
  - it is useful to write functions that handle this problem automatically

"Safe" read / write functions

- **readn( ) and writen( ) read and write exactly N bytes, if not incurring in errors or EOF**
- **readline( ) reads until it finds LF or fills the buffer, if not incurring in errors or EOF**
- **reference: UNP, fig. 3.14, 3.15, 3.16**
- **note: readline( ) must necessarely read bytes one by one, but uses another (private) function to fill a buffer more efficiently**



readn, writen, readline

- **the functions with capitalised initial automatically perform error control**

```
ssize_t readn (int fd, void *buf, size_t nbyte)
ssize_t Readn (int fd, void *buf, size_t nbyte)

ssize_t writen(int fd, const void *buf, size_t nbyte)
void Writen (int fd, const void *buf, size_t nbyte)

ssize_t readline (int fd, void *buf, size_t nbyte)
ssize_t Readline (int fd, void *buf, size_t nbyte)
```

sockwrap.h
sockwrap.c

Using errlib and sockwrap

- **sockwrap uses errlib to signal errors**
- **errlib requires that the variable “prog” exists and is initialised to the name of the program being executed**

```
#include "errlib.h"
#include “sockwrap.h"
char *prog;

int main (int argc, char *argv[])
{
  . . .
  prog = argv[0];
  . . .
}
```

Attention!

- **since my_read uses a local buffer, the readline functions aren’t re-entrant:**
  - they cannot be used in a multiprocess or multithread environment
  - for this cases, we should develop re-entrant functions that allocate externally the buffer for my_read
- **calls to readline cannot be intermixed with calls to normal read functions because they "read ahead" and therefore predate data that could be needed by a different read**

Example: TCP daytime client & server

- **the daytime service (tcp/13):**
  - provides current date and time in user friendly format
  - responses are terminated with CR+LF
- **develop a client connecting to the daytime server whose address is specified on the command line**
- **develop an (iterative) server that waits for service requests and provides date and time, identifying the connected client**

`daytimetcpc.c`
`daytimetcps.c`

Exercises

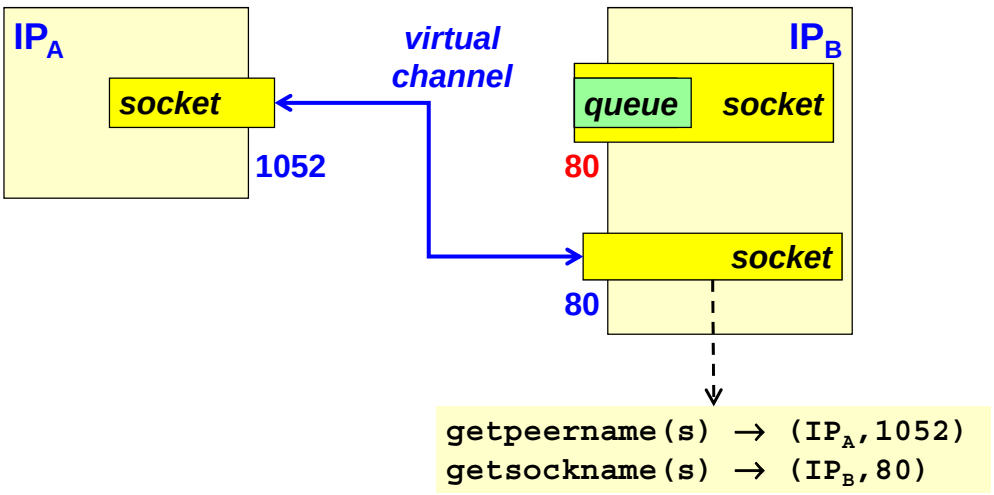
- **remove listen( ) from server:**
  - what happens? why?
- **leave listen( ) but remove bind( ):**
  - what happens? why?

Retrieving information on sockets

- **to know address and port**
  - of the local side: getsockname( )
  - of the remote side : getpeername( )
- **return 0 if OK, -1 in case of error**

```
#include <sys/socket.h>
int getsockname ( int sockfd,
  struct sockaddr *localaddr, socklen_t *addrp )
int getpeername ( int sockfd,
  struct sockaddr *peeraddr, socklen_t *addrp )
```

Information on sockets: example

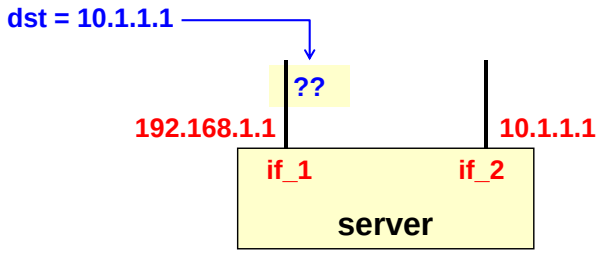


Implicit or explicit binding?

- **consider the following case:**
  - multi-homed web server (N network addresses)
  - one different site per each address
- **if server binds to INADDR_ANY:**
  - a single listening process
  - demultiplexing performed by the server
- **if servers binds to each of the N addresses:**
  - N listening processes
  - demultiplexing performed by the TCP/IP stack

weak-end / strong-end model

- **in case of multi-homed server …**
- **“strong end model” = kernels that accept packets on an interface only if DST_IP equals to interface’s IP**
- **“weak end model” = kernels that accept packets only if DST_IP equals to an IP of any server interface**



Porte dedicated to servers and clients

- **IANA**
  - 1-1023 = well-known ports
  - 1024-49151 = registered ports
  - 49152-65535 = dynamic / ephemeral ports
- **UNIX**
  - 1-1023 = reserved to processes with EUID=0
  - 513-1023 = reserved to privileged clients (r-cmds)
  - 1024-5000 = BSD ephemeral ports (few!)
  - 5001-65535 = BSD servers (non-privileged)
  - 32768-65535 = Solaris ephemeral ports
- **Windows does not perform any control**

The port “zero”

- **the port 0 is reserved and cannot be used for any TCP or UDP connection**
- **in the Unix socket interface the value zero can be used to request a random dynamic port to the OS:**
  - usually useful only with UDP

recvfrom( ) and sendto( )

- **used for datagram sockets, although they are usable also for stream sockets**
- **returns the number of bytes that have been read / written, -1 in case of error**
- **the "flags" parameter is usually zero (more details follow)**

```
#include <sys/socket.h>
int recvfrom ( int sockfd,
  void *buf, size_t nbytes, int flags,
  struct sockaddr *from, socklen_t *addrlenp )
int sendto ( int sockfd,
  const void *buf, size_t nbytes, int flags,
  const struct sockaddr *to, socklen_t addrlen )
```

Receiving data with recvfrom( ) (*)

- **receiving zero data is OK (=empty UDP payload) and does not indicates EOF (it does not exist with datagram sockets!)**
- **the "from" argument upon return tells us who sent the datagram**
  - similar to the one returned by accept()
- **using NULL as the "from" parameter indicates that you are not interested in knowing the protocol address of who sent the data**
  - … but then you don't know who sent the datagram and whom to respond to (!), apart receiving this information at application level

Binding with datagram sockets (*)

- **usually the client does not call bind( )**
  - the kernel assigns automatically a port to the socket the first time it is used; i.e. when calling sendto()
  - the kernel assigns automatically an address to the socket based on the outgoing interface; i.e. independently for every sent packet
- **alternatively the client can call bind( ) to the port 0 that instruct the kernel to assign an ephemeral random port**

Example: UDP daytime client & server

- **the daytime service (udp/13) provides current date and time in user friendly format**
- **the server sends date and time in a UDP packet to every client sending any packet (even empty) to it**
- **develop a client that requests date and time to the daytime server specified on the command line and prints the returned data**
- **develop a (iterative) server that waits for service requests and provides date and time, while identifying the requesting client**

`daytimeudpc.c`
`daytimeudps.c`

cygwin: known problems

- **sendto( ) with a zero sized buffer does not send anything (it should send UDP with a zero-length payload)**

Problems with datagram sockets

- **problem 1: since UDP is not reliable, the client risks to block indefinitely on reception, i.e. on recvfrom()**
- **using timeouts not always solves the problem:**
  - OK if resending the request is not a problem
  - unacceptable otherwise (e.g. debit or credit transaction)
- **problem 2: how to verify that the response actually comes from the server we send the request to?**
- **need to filter the responses either at user or at kernel level**

Problem 1

```
enum { normal, clock_rung } state;

clock (...) {
  state = clock_rung;
}

signal (SIGALRM, clock);
do {
  sendto (...);
  state = normal; alarm (timeout);
  recvfrom (...);
  alarm (0);
} while (state == clock_rung);
```

Problem 1: notes

- the solution with alarm is subject to a race condition (data arriving simultaneously with expiration of the timeout)
- with a “long” timeout, the probability is low but it cannot be excluded
- it is better to set a timeout directly on the socket with the SO_RCVTIMEO option or to use select( )

Verification of datagram responses

- binary comparison between the responder's address and the original destination's one
- possible problems with multi-homed servers
- solution 1: verify not the address but the DNS name (works only with registered servers)
- solution 2: the server explicitly binds to all its addresses (bind) and then waits on them all (select)

```
n = recvfrom(sfd,*buf,nbyte,0,(SA*)&from,&fromlen);
if ( (fromlen == serverlen) &&
     (memcmp(&servaddr,&from,serverlen) == 0) )
  // accepted response
```

Asynchronous errors

- when an error occurs to a UDP transmissions (e.g. port unreachable), an ICMP error packet is generated
- … but the sendto( ) function has already terminated with OK status
- … and therefore the kernel does not know which application the error should be forwarded to (and in which way!)
- possible solutions:
  - use connected (!) datagram sockets
  - intercept ICMP errors with an ad-hoc daemon

Connected datagram sockets

- **it is possible to call connect( ) on a datagram socket to specify once for all the intended peer**
- **consequences:**
  - can use write( ) or send( ) instead of sendto()
  - can use read() or recv() instead of recvfrom()
  - asynchronous errors are forwarded to the process controlling the socket
  - the kernel automatically performs response filtering and accepts only packets originating form the peer
- **in case of repeated communication with the same peer, it leads to fair performance improvement**

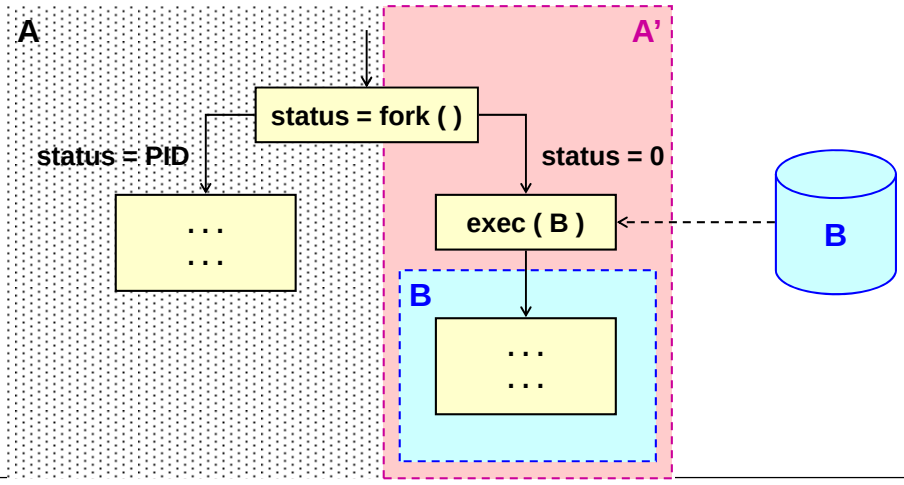
Disconnecting a datagram socket

- **it is possible to perform a second connect( ) towards another address to change the connection**
- **prohibited for stream sockets**
- **to disconnect completely (i.e. return back to a non-connected socket), call connect( ) towards an unspecified address using AF_UNSPEC**
- **in this case, you will receive an EAFNOSUPPORT error which can be safely ignored**

Concurrent Servers

Processes in Unix

- duplication of the application: fork( )
- possible execution of a new image: exec( )



fork( )

- **generates a new process …**
- **… which shares image and execution environment with the process that called fork( )**
- **return status:**
  - -1 in case of error
  - 0 to the new process, or “child”
  - the PID of the new process to the “parent”

```
#include <unistd.h>
pid_t fork (void);
```

getpid( ), getppid( )

- **child can retrieve the parent’s PID using getppid( )**
- **note: the parent is unique for every process**
- **to retrieve the process’ own PID use getpid( )**
- **return status:**
  - -1 in case of error
  - the desired PID

```
#include <unistd.h>
pid_t getpid (void);
pid_t getppid (void);
```

exec functions

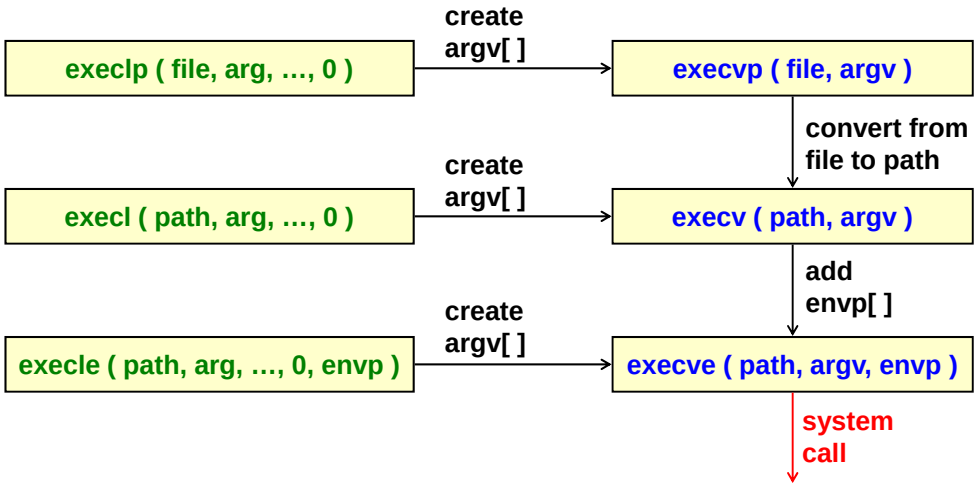
- replace the image being executed with a new image
- return -1 in case of error
- L functions pass arguments as a list of variables, terminated by NULL
- V functions pass arguments as an array of pointers, with NULL as the last element
- P functions locate the image based on PATH, the others require the full pathname
- E functions receive environment variables as an array of pointers, with a last NULL element; the others use the external variable “environ”

Functions execv( ) and execl( ) (*)

```
#include <unistd.h>
int execv ( const char *filename,
  char *const argv[] );
int execve ( const char *filename,
  char *const argv[], char *const envp[] );
int execvp ( const char *pathname,
  char *const argv[] );
int execl ( const char *filename,
  const char *arg0, ..., (char *)NULL );
int execle ( const char *filename,
  const char *arg0, ..., (char *)NULL,
  char *const envp[] );
int execlp ( const char *pathname,
  const char *arg0, ..., (char *)NULL );
```

exec functions

- **usually, only execve( ) is a system call**



Communicating through exec( )

- **explicit parameter passing:**
  - through the arguments
  - through the environment variables
- **file descriptors (files and sockets) remain open, if the caller does not use fcntl( ) to set the flag FD_CLOEXEC which makes them close when executing an exec**

Concurrent server skeleton (I)

```
pid_t pid; // child PID
int listenfd; // listening socket
int connfd; // communication socket

// create the listening socket
listenfd = Socket( ... );
servaddr = ...
Bind (listenfd, (SA*)&servaddr, sizeof(servaddr));
Listen (listenfd, LISTENQ);
```

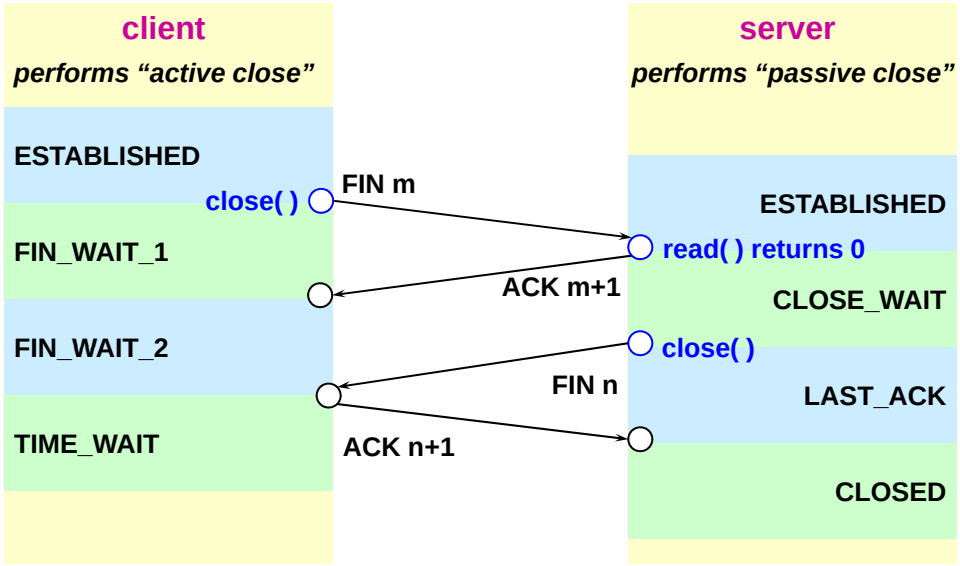
Concurrent server skeleton (II)

```
// server execution loop
while (1)
{
  connfd = Accept (listenfd, ...);
  if ( (pid = Fork()) == 0 )
  {
    Close(listenfd);
    doit(connfd); // the child performs its work
    Close(connfd);
    exit(0);
  }
  Close (connfd);
}
```

The importance of calling close( )

- **if the parent forgets to close the connection socket …**
  - it rapidly exhausts the descriptors
  - the channel with the client remains open even when the child has terminated and has closed the connection socket
- **notes:**
  - the close( ) function does not close the socket but it just decrements its reference count
  - only when REFCNT becomes zero, the socket is actually closed (i.e. FIN is sent in case of TCP sockets)

Closing connections



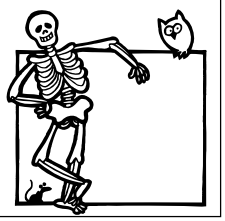
The state TIME_WAIT

- **the state TIME_WAIT is exited only by timeout:**
  - duration equal to 2 x MSL (Max Segment Lifetime)
  - MSL = 2 minutes (RFC-1122), 30 seconds (BSD)
  - therefore timeout = 1…4 minutes
- **it exists to solve two problems:**
  - to implement the full-duplex closure of TCP
    - the last ACK may get lost and the client receive a new FIN
  - to allow expiration of duplicated packets
    - they may be interpreted as part of a new incarnation of the same connection

Child termination

- **when a child process terminates, the SIGCHLD signal is sent to the parent**
- **default reaction:**
  - ignored
  - … which generates a “zombie” process
- **zombies are inherited and eliminated by the init process only when the parent process terminates (but usually servers never terminate …)**

If we want to avoid zombies,
we have to wait for our children.
-- W.R.Stevens



wait( ) and waitpid( )

- **return:**
  - the child PID; 0 or -1 in case of error
  - the termination status of the child
- **wait( ) is blocking**
- **waitpid( ):**
  - not blocking if using the option WNOHANG
  - allows specifying the PID of a particular child (-1 to wait for the first that terminates)

```
#include <sys/wait.h>
pid_t wait (int *status);
pid_t waitpid (pid_t pid, int *status, int options);
```

Intercepting SIGCHLD

- **if more children terminate "simultaneously" only one SIGCHLD is generated, thus you must wait them all**

```
#include <sys/wait.h>
void sigchld_h (int signum)
{
  pid_t pid;
  int status;
  while ( (pid = waitpid(-1,&status,WNOHANG)) > 0)
#ifdef TRACE
    err_msg (
"(%s) info - figlio %d terminato con status %d\n",
      prog, pid, status)
#endif
  ; // pay attention to this column
}
```

Interrupted system calls

- **when a process executes a “slow” system call (i.e. one that may block the caller)**
- **… it may unblock not because the system call has terminated**
- **… but because a signal has arrived; this case is characterised by errno == EINTR (or ECHILD)**
- **you should then handle this case and repeat the system call (this is already performed by sockwrap functions)**
- **it is a very relevant case with accept( ) in servers**
- **ATTENTION: you can repeat every system call but connect( ); in this case you should use select( ) to wait for the connection to complete**

Concurrent server: example

- **develop a concurrent server listening on port tcp/9999 that receives text lines containing two integers and returns their sum**
- **develop a client that:**
  - reads text lines from standard input
  - sends them to the port tcp/9999 of the server specified on the command line
  - receives the response lines and prints them on standard output

`addtcps.c`
`addtcpc.c`

Application robustness

Slow servers

- **if the server is slow or overloaded …**
- **… the client can complete the 3-way handshake and then terminate the connection (RST)**
- **… within the time the server employs between listen and accept**
- **this problem**
  - may be treated directly by the kernel or may generate the EPROTO / ECONNABORTED errors in accept
  - frequent case in overloaded web servers

Child-server termination

- **when the child-server communicating with the client terminates properly (exit), the socket is closed, i.e.:**
  - a FIN is generated, accepted by the client kernel but not transmitted to the application until the next read
  - if the client performs a write it receives a RST
  - depending on timing, the client will receive error on the write, or EOF or ECONNRESET on the next read
  - if the client has not yet read the data sent by the server before the channel is closed, this data may get lost (therefore it is better a partial closure on the server side with shutdown_write + timeout + exit)

SIGPIPE

- **a process calling write( ) on a socket that has received RST, receives the SIGPIPE signal**
  - default: terminates the process
  - if the signal is intercepted or ignored, write( ) returns the EPIPE error
- **attention:**
  - if there are many sockets open for writing …
  - … SIGPIPE does not indicate which one has generated the error
  - therefore, it is better to ignore the signal and handle the EPIPE returned by the write( )

Server crash

- **covers also the case of unreachable server**
- **client's writes will work (there is no one responding with an error!)**
- **client's reads will timeout (sometimes after long time: in BSD 9 m!) generating ETIMEDOUT**
- **reads and writes may receive EHOSTUNREACH or ENETUNREACH if an intermediate router has detected the problem and signalled it with ICMP**
- **solution: set a timeout**
  - directly on the socket with specific socket options
  - using select( )
  - using alarm( ) – deprecated

Server crash and reboot

- **sequence:**
  - crash (= unreachable server)
  - boot (=server is reachable but has lost knowledge of existing connections: RST)
- **as a consequence read and write will fail with ECONNRESET**

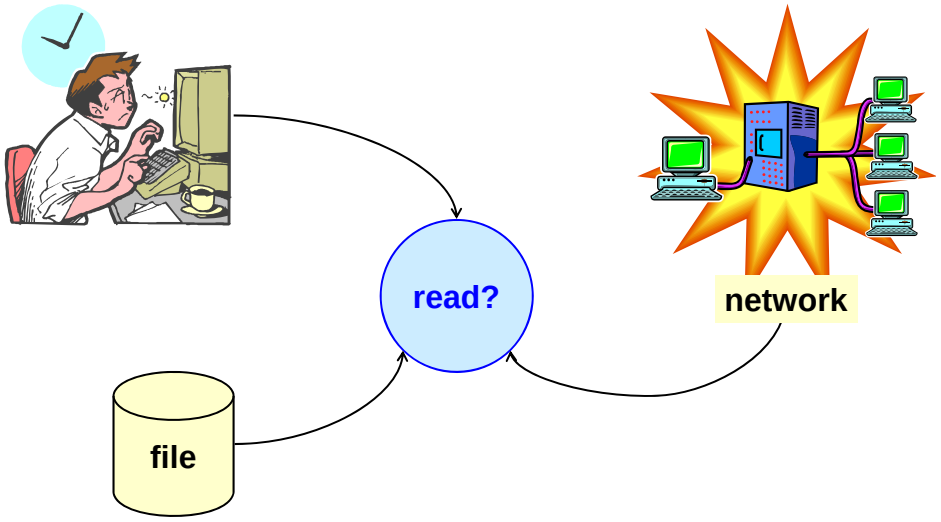
Server shutdown

- **at shutdown of a Unix node, the init process:**
  - sends SIGTERM to every active process
  - after 5…20 seconds sends SIGKILL
- **SIGTERM can be intercepted and thus a server can try to close all its open sockets**
- **SIGKILL cannot be intercepted; it terminates all the processes closing all their open sockets**

Heartbeating

- **if you want to know as soon as possible if the peer is unreachable or broken, you must activate an "heartbeating" mechanism**
- **two possible implementations:**
  - through the option SO_KEEPALIVE
  - through an application protocol for heartbeating

I/O multiplexing



Use of I/O multiplexing

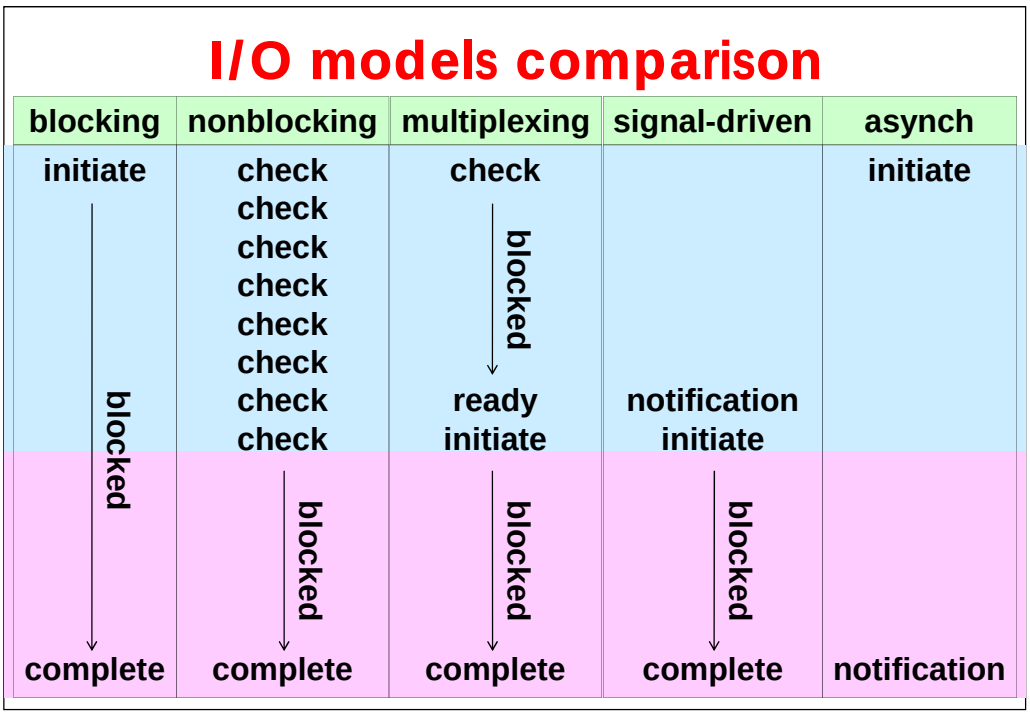
- a client managing inputs form multiple sources (typically the user on the keyboard and the server on a socket)
- a client managing multiple sockets (rare, but it's typical for web browsers)
- a TCP server managing both the listening socket and the connection ones (without activating separate processes; e.g. embedded systems)
- a server managing both UDP and TCP
- a server managing multiple services and/or protocols (rare, but typical for the inetd process)

I/O models

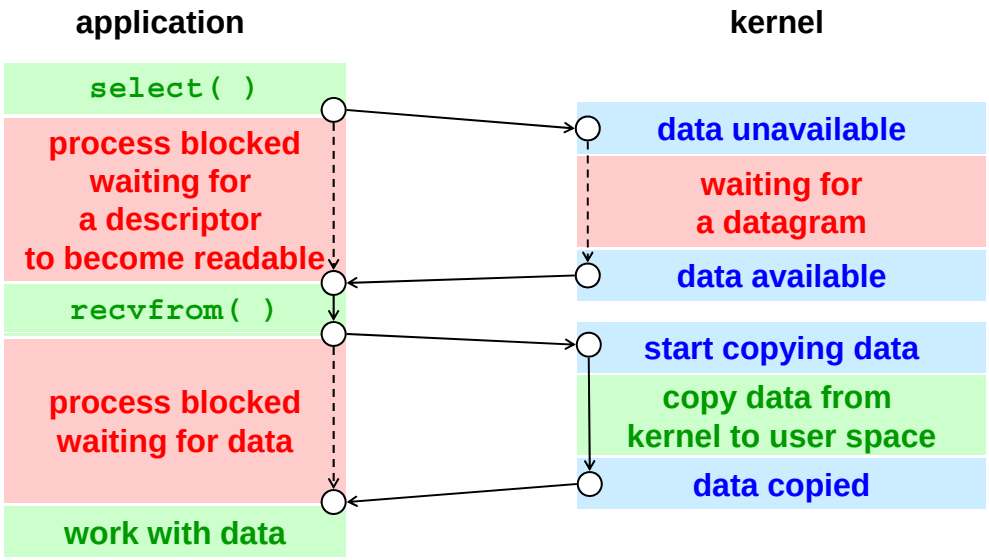
- **blocking (read on normal sockets)**
- **nonblocking (read on non-blocking sockets)**
- **multiplexing (select, poll)**
- **signal-driven (SIGIO)**
- **asynchronous (aio_xxx functions)**

- **all models encompass two phases:**
  - waiting for data to be ready
  - copying data from kernel space to user space
- **the problem is always on reading, almost never on writing**

I/O models comparison



I/O multiplexing model



select( )

- **blocks the calling process until …**
  - one of the selected descriptors becomes “active”
  - or the timeout expires (if set)
- **maxfdp1 (= MAX FD Plus 1) sets the number of the higher file descriptor to be monitored, plus one**
- **returns the number of active descriptors, 0 if terminated by timeout, -1 in case of error**

```
#include <sys/select.h>
#include <sys/time.h>
int select (int maxfdp1, fd_set *readset,
  fd_set *writeset, fd_set *exceptset,
  struct timeval *timeout);
```

Timeout

- unlimited waiting: timeout == NULL
- maximum waiting time: timeout != NULL
- no waiting (i.e. polling): timeout == { 0, 0 }
- some systems modify the timeout value, therefore it is better to re-initialize it at every call

```
#include <sys/time.h>
struct timeval
{
  long tv_sec;  // seconds
  long tv_usec; // microseconds
};
```

fd_set

- set of flags to select file descriptors (i.e. a “bit mask”)
- manipulated with the macros FD_xxx
- use FD_ISSET to discover on which descriptors there was some activity
- attention!!! they must be re-initialized at every call

```
#include <sys/select.h>
void FD_ZERO (fd_set *fdset); // resets the mask
void FD_SET (int fd, fd_set *fdset); // set(fd)
void FD_CLR (int fd, fd_set *fdset); // reset(fd)
int  FD_ISSET (int fd, fd_set *fdset); // test(fd)
```

When a descriptor is “ready”?

- **readset:**
  - there is data to read
  - the peer has closed the channel for read (i.e. EOF)
  - an error occurred on the descriptor
  - there is a new connection for a listening socket
- **writeset:**
  - there is space available for writing
  - peer closed the channel for writing (SIGPIPE/EPIPE)
  - an error occurred on the descriptor
- **exceptset:**
  - there is OOB data available

I/O multiplexing: example

- **modify the client addtcpc.c to maximise throughput**
- **solution:**
  - do not alternate between reading the operation from standard input and reading the response from the socket but manage both by using select( )
  - see diagrams in the next slides
- **note: perform the test by typing input on the keyboard, and by redirecting it from a file … you will note an error!**

`addtcpc2.c`

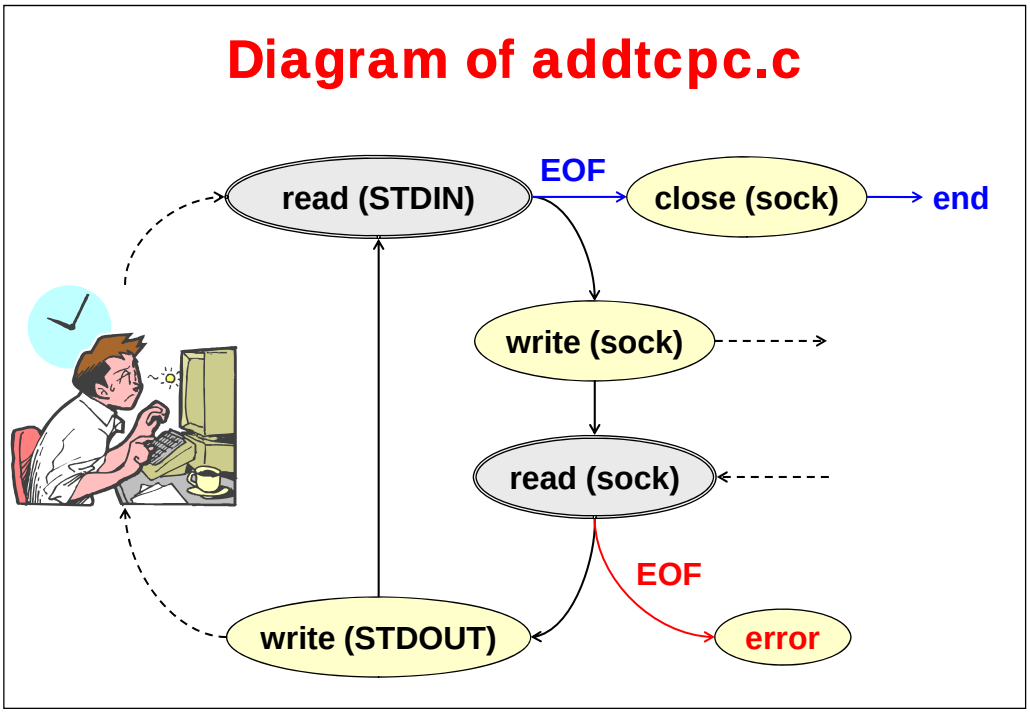


Diagram of addtcpc.c
read (STDIN)
EOF
close (sock)
end
write (sock)
read (sock)
EOF
write (STDOUT)
error

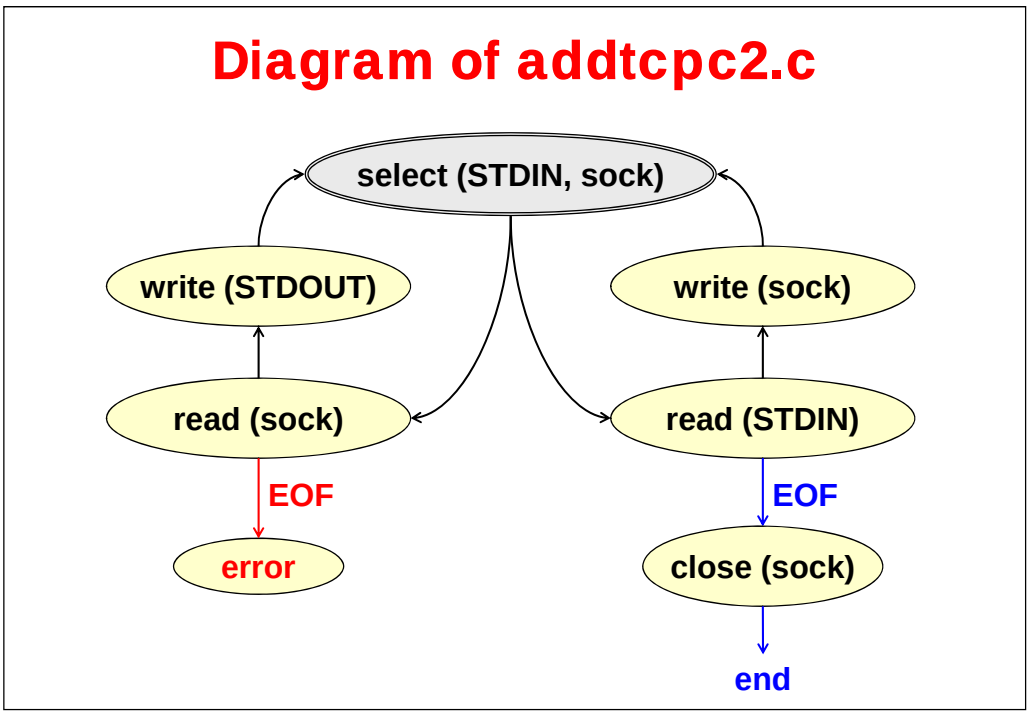


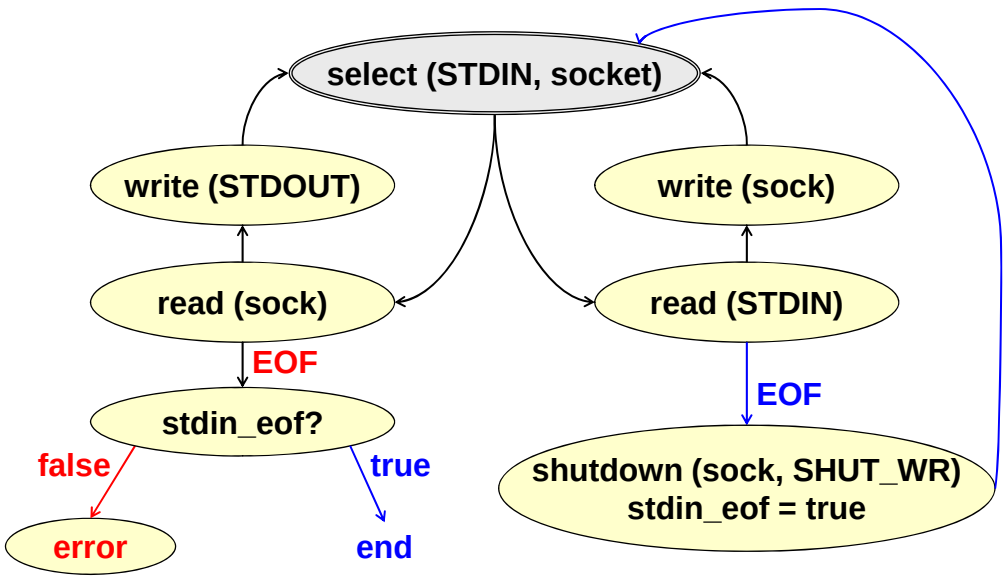
Diagram of addtcpc2.c
select (STDIN, sock)
write (STDOUT)
write (sock)
read (sock)
read (STDIN)
EOF
EOF
error
close (sock)
end

Batch input

- **when input is fed without interruptions (like in case of reading from a file with a large buffer), it may happen to terminate all the input and close the socket without waiting for all the responses**
- **solution:**
  - do not close the socket completely, i.e. by calling close()
  - but instead close only the writing part, by calling shutdown()
  - and wait for EOF (from server) before closing the reading part

`addtcpc3.c`

Scheme of addtcpc3.c



shutdown( )

- **close one of the two channels associated to a socket**
- **note that close( ):**
  - closes both channels (… but only if the reference count of the descriptor becomes 0)
  - the exact behaviour depends on the option LINGER
- **possible values for the 'howto' parameter:**
  - SHUT_RD (or 0)
  - SHUT_WR (or 1)
  - SHUT_RDWR (or 2)

```
#include <sys/socket.h>
int shutdown (int sockfd, int howto);
```

Behaviour of shutdown( )

- **shutdown (sd, SHUT_RD)**
  - read on the socket is impossible
  - the content of the read buffer is discarded
  - future data will be discarded directly by the stack
- **shutdown (sd, SHUT_WR)**
  - write on the socket is impossible
  - the content of the write buffer is sent to the destination, followed by FIN for TCP sockets

Socket options

getsockopt( ) and setsockopt( )

- applicable only to open sockets
- "level" indentifies the level of the network stack: SOL_SOCKET, IPPROTO_IP, IPPROTO_TCP, ...
- mnemonic values for "optname"

```
#include <sys/socket.h>
#include <netinet/tcp.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);
```

Some options at SOCKET level

level	optname	get	set	type
SOL_SOCKET	SO_BROADCAST	X	X	int (boolean)
	SO_DEBUG	X	X	int (boolean)
	SO_DONTROUTE	X	X	int (boolean)
	SO_ERROR	X		int
	SO_KEEPALIVE	X	X	int (boolean)
	SO_LINGER	X	X	struct linger
	SO_OOBINLINE	X	X	int (boolean)
	SO_RCVBUF	X	X	int
	SO_SNDBUF	X	X	int
	SO_RCVTIMEO	X	X	struct timeval
	SO_SNDTIMEO	X	X	struct timeval
	SO_REUSEADDR	X	X	int (boolean)
	SO_REUSEPORT	X	X	int (boolean)
	SO_TYPE	X		int

Some options at IP and TCP level

level	optname	get	set	type
IPPROTO_IP	IP_OPTIONS	X	X	
	IP_TOS	X	X	int
	IP_TTL	X	X	int
	IP_RECVDSTADDR	X	X	int
IPPROTO_TCP	TCP_MAXSEG	X	X	int
	TCP_NODELAY	X	X	int
	TCP_KEEPALIVE	X	X	int

Example of reading socket options

- **from UNP, section 7.3**
- **note: in CYGWIN the timeouts are integers**

`checkopts.c`

Broadcast, Keepalive, buffer

- **SO_BROADCAST**
  - applicable only to datagram socket
  - enables the use of broadcast addresses
- **SO_KEEPALIVE**
  - exchange a “probe” packet every 2 hours (!)
  - configuration at kernel level to change the value
- **SO_SNDBUF, SO_RCVBUF**
  - sizes of the local buffers; set before connect (for clients) and listen (for servers)
  - value >= 4 x MSS
  - value >= bandwidth x RTT

SO_SNDTIMEO, SO_RCVTIMEO

- **POSIX specifies timeouts with a timeval struct**
- **originally it was just an integer**
- **they apply only to:**
  - read, readv, recv, recvfrom, recvmsg
  - write, writev, send, sendto, sendmsg
- **use other techniques for accept, connect, …**

```
#include <sys/time.h>
struct timeval
{
  long tv_sec;
  long tv_usec;
};
```

SO_REUSEADDR (SO_REUSEPORT)

- **SO_REUSEADDR allows:**
  - binding to a local port occupied by a process (a connection and a listening socket)
  - having multiple processes on the same port (e.g. IP_A, IP_B, INADDR_ANY)
  - having multiple sockets of a single process on the same port but with different local addresses (useful fro UDP without IP_RECVDSTADDDR)
  - having multiple multicast sockets on the same port (some systems use SO_REUSEPORT)
- **option highly recommended for all TCP servers**

SO_LINGER

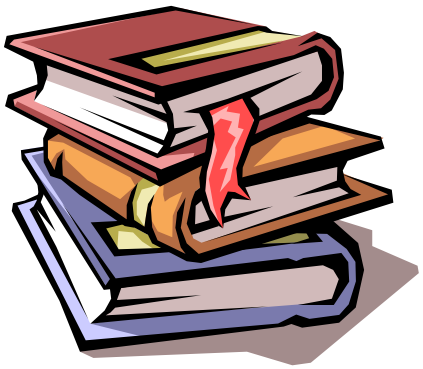
- changes the behaviour of close( )
- OFF: non-blocking close (but try to send pending data)
- ON + l_linger == 0: non-blocking close and connection abort (=RST, instead of FIN + TIME_WAIT)
- ON + l_linger > 0: blocking close (try to send all pending data until the timeout; if the timeout expires, returns EWOULDBLOCK)

```
#include <sys/socket.h>
struct linger
{
  int l_onoff; // 0=off, non-zero=on
  int l_linger; // linger timeout (seconds in Posix)
};
```

IP_TOS, IP_TTL

- read or set the TOS and TTL values of sent packets
- values for TOS:
  - IPTOS_LOWDELAY
  - IPTOS_THROUGHPUT
  - IPTOS_RELIABILITY
  - IPTOS_LOWCOST (IPTOS_MINCOST)
- values that should be the default for TTL:
  - 64 for UDP and TCP (RFC-1700)
  - 255 for RAW sockets

Supporting libraries



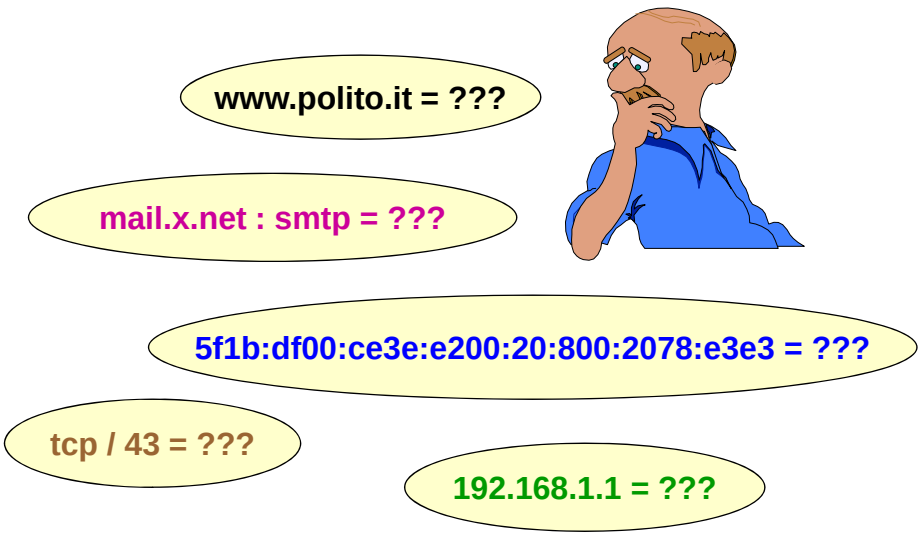
Network libraries

- **libpcap**
  - packet capture
  - http://www.tcpdump.org
  - only for Unix; for Windows see winpcap
- **libdnet**
  - raw IP, raw Ethernet, arp, route, fw, if, addresses
  - http://libdnet.sourceforge.net
  - for Unix and Windows
- **libnet**
  - http://www.packetfactory.net/projects/libnet/

Event libraries

- **libevent**
  - http://monkey.org/~provos/libevent/
- **liboop**
  - http://liboop.org/

Managing names and addresses



Mnemonic and numerical formats

- **conversion between mnemonic names and numerical values (nodes, services, networks, protocols, network addresses)**
- **attention! conversion depends on local system configuration**
  - local files (e.g. /etc/hosts, /etc/services)
  - LAN lookup services (e.g. NIS, LDAP)
  - global lookup service (DNS)
- **only for DNS you can (with some efforts) point explicitly to a specific service**

gethostbyname( )

- **returns a data structure with the description of the node whose name is specified as argument**
- **in case of error, it returns NULL and sets the variable h_errno to specify the error, whose textual representation can be retrieved with hstrerror(h_errno)**

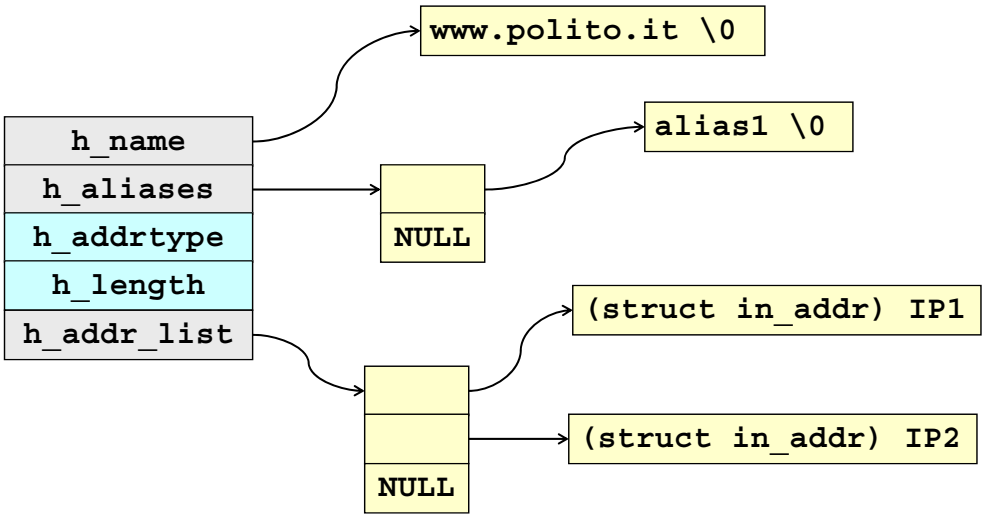
```
#include <netdb.h>
struct hostent *gethostbyname (
  const char *hostname );
extern int h_errno;
char *hstrerror (int h_errno);
```

struct hostent

- **note: addresses represented as 1 char = 1 byte (better: 1 unsigned char)**

```
#include <netdb.h>
struct hostent {
  char *h_name; // canonical name
  char **h_aliases; // array of alias names
  int  h_addr_type; // AF_INET or AF_INET6
  int  h_length; // address length (4 or 16 bytes)
  char **h_addr_list; // array of addresses
};
#define h_addr h_addr_list[0] // BSD compatibility
```

struct hostent



gethostbyaddr( )

- returns a data structure with the description of the node whose address is specified as argument
- in case of error, it returns NULL and sets the variable h_errno to specify the error, whose textual representation can be retrieved with hstrerror(h_errno)
- note: the argument “addr” is actually a pointer to the struct in_addr or in_addr6

```
#include <netdb.h>
struct hostent *gethostbyaddr (
  const char *addr, size_t len, int family );
```

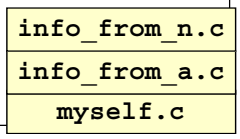
uname()

- identifies the node where the process is executing
- strings’ size and content depend on the system and its configuration
- in case of error, it returns a negative integer and sets the errno variable

```
#include <sys/utsname.h>
struct utsname {
  char sysname[...];  // OS name
  char nodename[...]; // network node name
  char release[...];  // OS release
  char version[...];  // OS version
  char machine[...];  // CPU type
};
int uname (struct utsname *nameptr);
```

Examples

- (info_from_n.c) program to provide all information available on a node given its name
- (info_from_a.c) program to provide all information available on a node given its address
- (myself.c) program to provide all information available on the node where the program is executing



getservbyname( ), getservbyport( )

- return information on the service whose name or port is passed as argument
- return NULL in case of error
- attention: the returned port numbers use network order (OK to program, not to visualise them)

```
#include <netdb.h>
struct servent *getservbyname (
  const char *servname, const char *protoname );
struct servent *getservbyport (
  int port, const char *protoname );
```

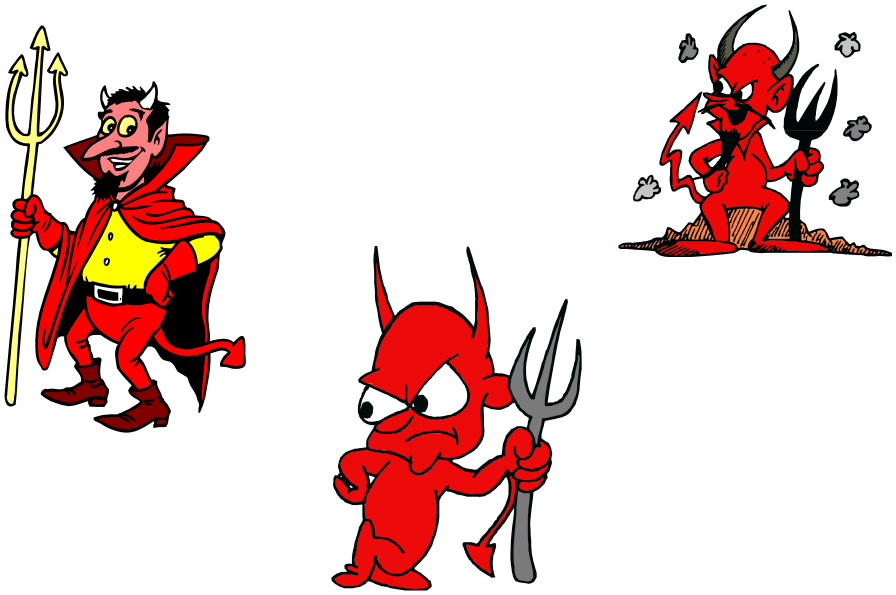
struct servent

- note: error in UNP (p.251) to declare int s_port

```
#include <netdb.h>
struct servent {
  char  *s_name; // official service name
  char **s_aliases; // array of aliases
  short  s_port; // port number (network order)
  char  *s_proto; // transport protocol
};
```

service.c

The “daemons”



What is a daemon?

- an autonomous process
- typically plays the server role
- automatically started at boot time
- detached by any terminal
- writes information to a log file
- configured through information passed on the command line (deprecated!) or contained in a configuration file
- executed with the rights of a given user and group
- works within a given directory

syslog( ) (*)

- mandatory for Unix98; POSIX standardises less features than what available in most UNIX versions
- allows to append data to the system log
- the actual destination of data depends on the syslogd configuration (e.g. /etc/syslog.conf)
- communication is opened implicitly or explicitly (openlog + closelog)

```
#include <syslog.h>
void syslog (
  int priority, const char *message , ...);
void openlog (
  const char *ident, int options, int facility);
void closelog (void);
```

syslog priority = level | facility

LEVEL

`LOG_EMERG`	(maximum priority)
`LOG_ALERT`	
`LOG_CRIT`	
`LOG_ERR`	
`LOG_WARNING`	
`LOG_NOTICE`	(default priority)
`LOG_INFO`	
`LOG_DEBUG`	(minimum priority)

FACILITY

`LOG_AUTH`	`LOG_AUTHPRIV`	`LOG_CRON`
`LOG_DAEMON`	`LOG_FTP`	`LOG_KERN`
`LOG_LOCAL0`	`. . .`	`LOG_LOCAL7`
`LOG_LPR`	`LOG_MAIL`	`LOG_NEWS`
`LOG_SYSLOG`	`LOG_USER`	`LOG_UUCP`

syslog options

OPTIONS

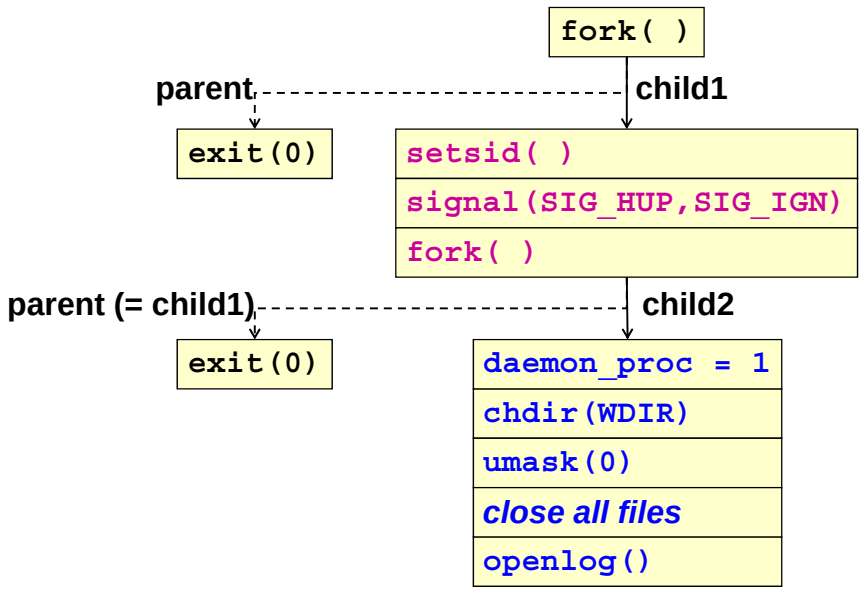
`LOG_CONS`	log on console if the communication with syslogd fails
`LOG_NDELAY`	immediate socket opening, without waiting the first call to syslog( )
`LOG_PERROR`	log also on stderr
`LOG_PID`	insert in the log also the PID

Daemon initialization

- **disassociate from the controlling terminal (thus becoming immune to HUP, INT, WINCH that can be used for other signalling)**
- **move to the working directory**
- **close all inherited files**
- **optionally, as a precaution against foreign libraries:**
  - open /dev/null and associate it to stdin, stdout, stderr
  - or open a log file and associate it to stdout e stderr
- **open syslog**

- **to avoid mistakes during daemon initialization, use the daemon_init( ) function (UNP, p.336)**

daemon_init( )



Signals to daemons

- **conventions frequently used:**
  - SIGHUP to make the daemon re-reads the configuration file
  - SIGINT to make the daemon terminates

inetd

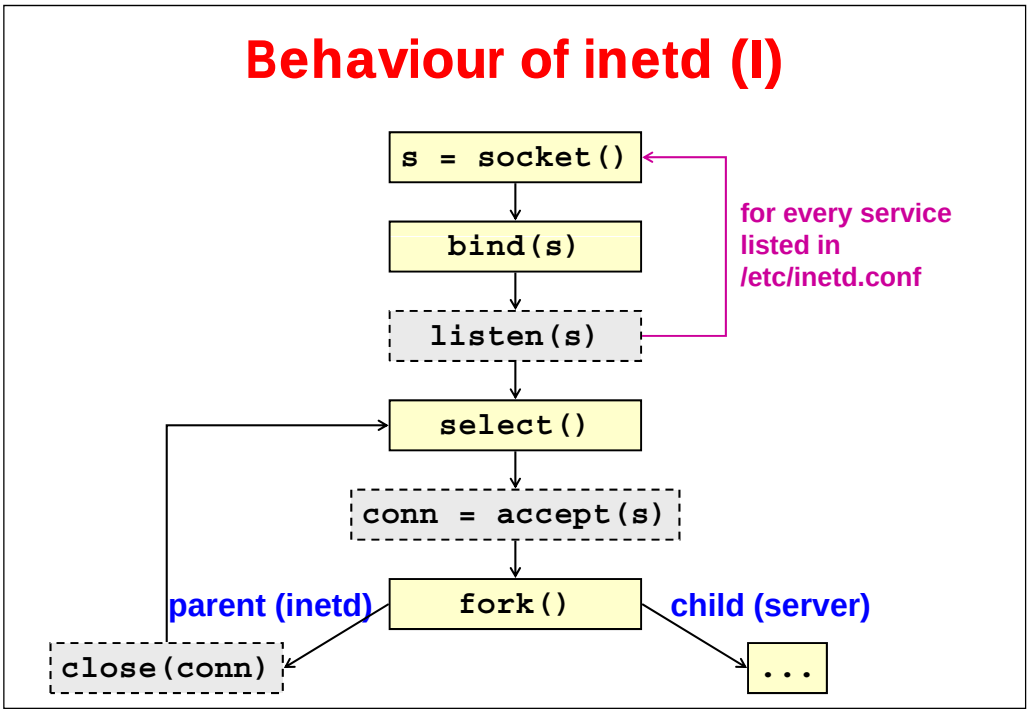
- **if a network node offers multiple services, it must have multiple listening daemons, everyone being a process and having some associated code**
- **for simplicity, Unix often uses the super-server "inetd"**
  - knows the services specified in /etc/inetd.conf
  - listens on all corresponding sockets
  - when it receives a connection request, it starts the corresponding server

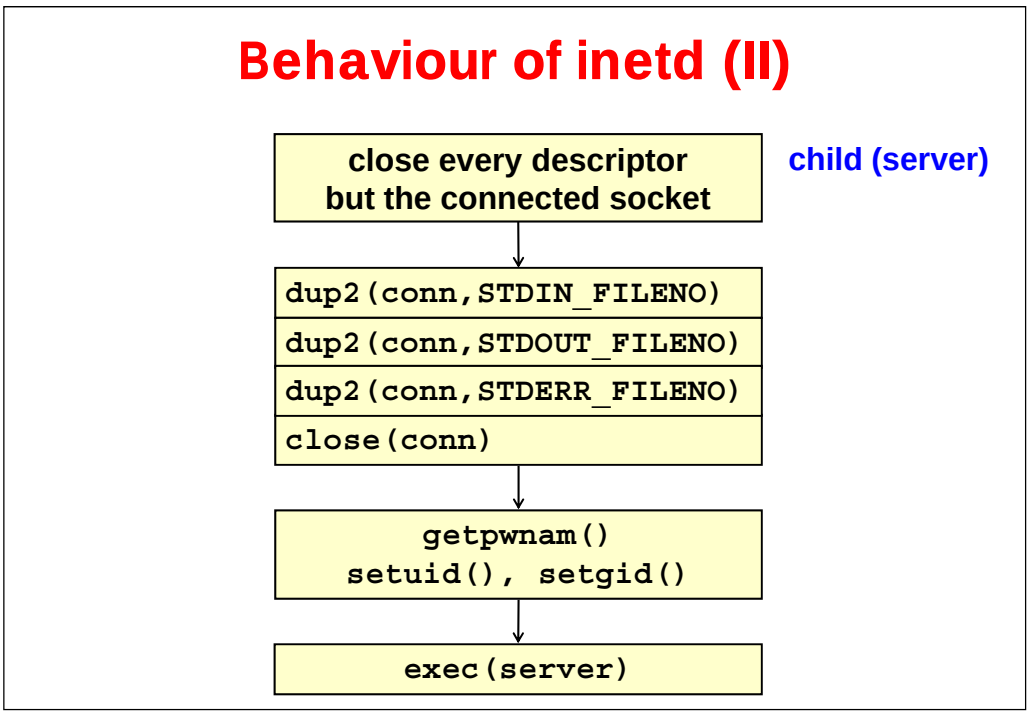
Format of /etc/inetd.conf

- **every line contains from 7 to 11 fields:**
  - service-name -- name of the service in **`/etc/services`**)
  - socket-type (**`stream`**, **`dgram`**)
  - protocol (**`tcp`**, **`udp`**)
  - wait-flag (**`wait`**, **`nowait`**) -- iterative or concurrent server
  - login-name (entry in **`/etc/passwd`**) -- user name
  - server-program (pathname, **`internal`**)
  - arguments (maximum 5, included **`argv[0]`**)

Example of /etc/inetd.conf

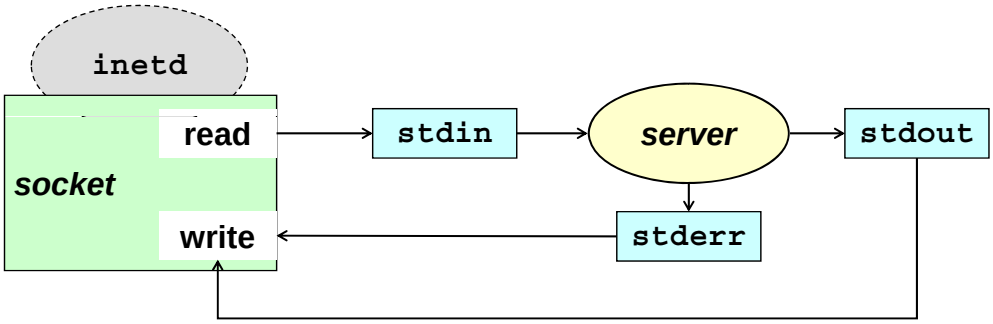
```
. . .
echo    stream  tcp  nowait  root    internal
echo    dgram   udp  wait    root    internal
ftp     stream  tcp  nowait  root    /usr/etc/ftpd ftpd
telnet  stream  tcp  nowait  root    /usr/etc/telnetd telnetd
login   stream  tcp  nowait  root    /etc/rlogind rlogind
tftp    dgram   udp  wait    nobody  /usr/etc/tftpd tftpd
talk    dgram   udp  wait    root    /etc/talkd talkd
spawn   stream  tcp  nowait  lioy    /usr/local/spawner spawner
. . .
```


Behaviour of inetd (I)
s = socket()
bind(s)
listen(s)
for every service
listed in
/etc/inetd.conf
select()
conn = accept(s)
parent (inetd)
fork()
child (server)
close(conn)
...


Behaviour of inetd (II)
close every descriptor
but the connected socket
child (server)
dup2(conn,STDIN_FILENO)
dup2(conn,STDOUT_FILENO)
dup2(conn,STDERR_FILENO)
close(conn)
getpwnam()
setuid(), setgid()
exec(server)

Server started by inetd

- **just need a "filter" that:**
  - reads data from stdin (descriptor 0)
  - generates results on stdout / stderr (descriptor 1 / 2)
- **problem with non-ASCI data**



Other solutions for the super-server

- **tcpd is a system to securely activate a server from inetd**
- **xinetd is an improved version of inetd that already includes the features of tcpd**
- **tcpserver is a full replacement of inetd+tcpd (but it is not a super-server)**

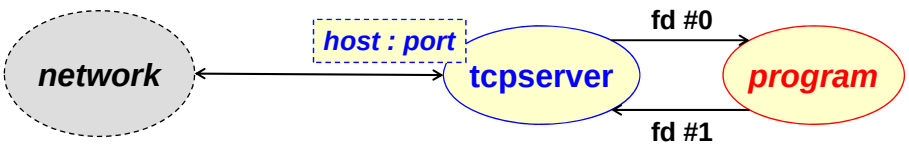
tcpserver

- **by D.J.Bernstein (http://cr.yp.to)**
- **is a filter, listening on a TCP port**
- **one copy for each service to protect**
  - independent copies = efficiency
  - small = easier to verify
- **flexibility, security and modularity**
- **meticulous attention to permissions and restrictions**
- **control on the number of concurrent processes**
- **access control (IP addresses, DNS names)**
- **configurable UID and GID**

tcpserver: activation

- **tcpserver waits listening on a port (port) of an interface (host) and executes an application (program) for each received request ...**
- **... if the optionally specified controls (-x) are satisfied**

```
tcpserver [ -qQvdDoOpPhHrR1 ] [ -climit ]
  [ -bbacklog ] [ -xrules.cdb ] [ -ggid ]
  [ -uuid ] [ -llocalname ][ -ttimeout ]
  host port program [ arg ...  ]
```



tcpserver: main options

- **-c*n* (maximum *n* simultaneous processes)**
- **-x*r.cdb* (control access rules in *r.cdb*)**
- **-ggid (set the group ID*)***
- **-u*uid* (set the user ID)**
- **-b*n* (allow a backlog of *n* TCP SYNs)**

tcpserver: access rules

- **rules defined according to the following format:**
  - user@addres:instructions_list
  - instructions_list ::= deny | allow , environment var.
- **rules compiled with tcprules, to obtain a file .CDB (hash data structure)**

```
tcprules rules.cdb rules.tmp < rules.txt
```

```
(rules.txt)
130.192.:allow
192.168:allow,CLIENT="private"
:deny
```