UNIX

Sockets

mgr inż. Marcin Borkowski

Introduction to Sockets

- Interprocess Communication channel:
  - descriptor based
  - two way communication
  - can connect processes on different machines
- Three most typical socket types (colloquial names):
  - **unix** (local connection)
  - **tcp** (reliable, byte stream connection over network)
  - **udp** (unreliable, datagram connection over network)

Introduction to Sockets

- Communication styles
  - *SOCK_STREAM*
    - binary stream
    - reliable
    - context
    - communication with one peer only
    - no records boundaries
    - connection oriented
    - fifo-like connection type
    - bidirectional
    - out of the band data may be (and usually is) supported

Introduction to Sockets

- Communication styles
  - *SOCK_DGRAM*
    - packets
    - <u>**data loss is possible**</u>
    - <u>**data duplicates are possible**</u>
    - <u>**order of delivery may be different than the order of sending**</u>
    - connectionless
    - communication with many peers
    - each packet has to be individually addressed

Introduction to Sockets

- Communication styles (cont.)
  - *SOCK_RAW*
    - low level network access
    - e.g. icmp ping request
    - requires proper capability or root privileges
    - SOCK_DGRAM similar type of connection

Introduction to Sockets

- Addresses, namespace, domain
  - usually **internet** *AF_INET* or **local** (**unix**) *AF_LOCAL, AF_UNIX*
  - new socket can can be bound to an address
    - obligatory for server address
    - client socket is usually bound automatically on first send or connect operation

Introduction to Sockets

- Protocol
  - **tcp** for byte stream contextual reliable connection
  - **udp** for packet contextless unreliable connection
  - usually programs use default (zero) protocol for given communication style and name space
- Connection requires both ends to use the same protocol in the same namespace and communication style

Introduction to Sockets

- Structure `sockaddr` is used for all namespaces:

  ```
  sa_family_t sa_family
  char sa_data[14]
  ```

  - this type is used only to cast addresses used as parameters to `bind`, `getsockname` and other functions
  - `sa_family` can be one of *AF_INET, AF_LOCAL, AF_UNIX*
  - each namespace defines its own structure, field `sa_family` is common to all those structures and can be used to recognize socket type

Introduction to Sockets

- If other end of stream socket connection is closed and the connection buffer is empty:
  - reading from this channel will return end of file status (read returns zero as number of bytes read)
  - writing to the channel results in delivery of SIGPIPE signal. If this signal is handled, ignored or blocked `write/send/sendto` will return the EPIPE error
  - writing and reading may result in ECONNRESET error
- EPIPE and ECONNRESET are not critical errors and can be properly recognized and handled in students' apps without terminating the application

Introduction to Sockets

- Socket connection typical use case (very rarely modified)
  - server (connection listener)
    - create socket
    - bind it with the address
    - set listen queue (SOCK_STREAM only)
    - accept connections (SOCK_STREAM only)
    - do work
    - close socket
  - client (connection initiator)
    - create socket
    - optionally bind with an address
    - connect
    - do work
    - close

Introduction to Sockets

- Out of Band data
  - priority messages can be send over the socket
  - usually used to indicate exceptional conditions
  - data is sent independently of the ordinary data
  - process must use `send/recv` functions with MSG_OOB flag, without this flag only ordinary data will be received
  - `select` and `poll` function can wait for OOB data
  - special marker is set to indicate the position of band data in the ordinary data
  - more information: *glibc manual chapter "Out-of-Band Data"*

Inter-Architectural Socket Connection

- Sockets can be used to communicate processes working on different architectures, this affects the way data types are treated
- **<u>Failure to recognize the problem is the most common source of errors in students' works</u>**
- Byte Order
  - does not affect 1 byte data types
    - `char` (not unicode `wchar_t`)
    - `[u]int8_t`
    - the safest method to send data is to format it as text !

Inter-Architectural Socket Connection

- Byte Order (cont.)
  - 2- and more- byte integers
    - little-endian less significant byte goes first
      - the higher memory address the more significant byte (255=FF00)
      - x86, amd64
    - big-endian more significant byte goes first
      - the lower memory address the more significant byte (255=00FF)
      - SPARC, PPC
    - network byte order (big-endian), <u>all data should be converted to network byte order before being sent and back to host byte order after being received</u>

Inter-Architectural Socket Connection

- Byte Order (cont.)
  - 2 byte data type ([u]int16_t) can be converted to network order with `htons` function and back to host order with `ntohs` function
  - 4 byte data type ([u]int32_t) can be converted to network order with `htonl` function and back to host order with `ntohl` function
  - 8 byte data types ([u]int64_t) must be converted manually, no standard function or macro exists, it is easy to write one though

Inter-Architectural Socket Connection

- Byte Order (cont.)
  - `double` and `float`
    - do not have network format defined !
    - the floating point format may be different !
    - it should be sent as human readable string and parsed at destination
  - arrays
    - all members of array should be converted separately
  - structures
    - all members of a structure should be converted separately
    - structure must be packed (see next slides)

Inter-Architectural Socket Connection

- Data types sizes
  - `char` is a one-byte data type (always)
  - do not use traditional data types `int` and `long` as they may differ in size
  - use UNIX standardized integer types:
    - `[u]int[8|16|32|64]_t`
- Structures have different field alignment
  - some architectures prefer 2,4 or 8 byte alignment
  - if communicating architectures use different alignment the structure will be malformed after transport

Inter-Architectural Socket Connection

- Structures have different field alignment (cont.)
  - compiler can enforce the smallest alignment possible that is the same on all architectures – so called *packing*
  - unfortunately packing options are specific to given compilers
    - some use `#pragma pack;`
    - gcc uses `__attribute__((__packed__))` on all structures to pack or `-fpack-struct` option to pack all structures in the code
  - **<u>there is no portable way of packing, structures should not be sent over socket in portable programs</u>**

Socket Functions

- Function `socket` :
  - creates unbound and unconnected socket descriptor
  - `domain` selects namespace, usually one of *PF_INET, PF_LOCAL, PF_UNIX* **(notice that constant has PF_ not AF_ prefix)**
  - `type` indicates communication style, usually SOCK_STREAM or SOCK_DGRAM
  - `protocol` usually defaults to zero as only one protocol exists for each combination of `domain` and `type`

Socket Functions

- Function `socketpair`:
  - creates a pair of unnamed and connected socket
  - only related processes can use it
  - can be used the same way as pipe, except that it offers two-way communication
  - `domain` can be only *PF_LOCAL, PF_UNIX* on Linux and most of other systems
  - other parameters are passed similarly as to socket function
  - `socket_vector` – array of 2 undistinguishable socket descriptors (return parameter)

Socket Functions

- Function `bind`:
  - assigns name to the socket
    - not obligatory, process can use unbound socket, and the name will be assigned automatically, process can use f. `getsockname` to learn this name
    - name of local end of the connection is assigned
    - name of remote peer can be read with f. `getpeername`
  - address must be cast to `struct sockaddr *`
  - address structure real length must be calculated accordingly to namespace type (see examples) as <u>it is not always matter of a simple</u> `sizeof`.

Socket Functions

- Functions `setsockopt` and `getsockopt`:
  - control various socket options (see examples)
  - useful POSIX compliant options for AF_INET:
    - SO_REUSEADDR(SOL_SOCKET level)
      - application fails to bind to an address some time after previous run (no mater how it was terminated). It takes even a few minutes for the system to clear the address, this option will speed up reuse, it should be instant but makes tcp less reliable as time-out for lost packets is removed
    - SO_KEEPALIVE(SOL_SOCKET level)
      - starts transparent exchange of test message, if connection is broken it can speed up detection of the problem. Waiting processes are sent SIGPIPE if test message fails to go through.

Socket Functions

- Functions `setsockopt` and `getsockopt`(cont.):
    - SO_BROADCAST (SOL_SOCKET level)
        - permits sending broadcasts form the sockets
        - broadcast can be blocked on firewall, usually no broadcasts form the external network are allowed in LAN
    - IP_MTU (IPPROTO_IP level)
        - reads current known MTU (Maximum Transport Unit) - the maximum reliable size of datagram process can send
        - <u>can be obtained (never set) for connected socket only</u>
        - depends on network connection
        - can change
        - cannot be less than 576
    - SO_ERROR (SOL_SOCKET level)
        - check for network pending errors

Socket Functions

- Function `connect` :
  - connects socket with the other end
  - usually client connects to the server
  - socket being connected does not need to be bound to any name
  - server socket must have a name

Socket Functions

- Function `connect` (cont.):
  - packet (datagram)
    - does not require permanent connection as each packet can be addressed individually
    - connecting is still possible and useful when many packets are to be sent to the same destination
    - process can connect the same packet socket many times to change destination address
    - connect will never block as the real connection is not established

Socket Functions

- Function `connect` (cont.):
  - SOCK_STREAM connection
    - connection is obligatory and possible only once per given socket
    - connect will block (provided that O_NONBLOCK flag was not set on socket descriptor) until connection is made
    - if socket is in non-blocking mode, or connect was interrupted by signal handling routine, connecting will **continue asynchronously** !!! In such a case:
      - **<u>process should wait for socket write readiness (see examples)</u>**
      - or assume socket to be connected, sleep 1-2 sec. If assumption is wrong the first operation on socket will fail (rather untidy approach)
      - <u>repeated connect returns EALREADY error, it must be recognised if TEMP_FAILURE_RETRY macro is used</u>

Socket Functions

- Function `listen`:
  - applies only to byte stream communication
  - enables connection requests on the socket
  - marks socket as server socket that should be used only to accept connections
  - `backlog` argument
    - merely a hint of size of queue of pending connections (i.e. connections already initiated by client and not yet accepted)
    - **not** the maximum number of simultaneously connected clients
    - POSIX suggests that passing zero as backlog parameter may mean implementation-defined minimal value
    - if pending connections queue is exhausted peer may be rejected

Socket Functions

- Function `accept`:
  - applies only to byte stream communication
  - creates a new socket for communication with connecting peer, returns new descriptor
  - new socket communication type and namespace are inherited from listening socket
  - newly created socket <u>may</u> not inherit flags set on listening socket (on Linux flags are not inherited), portable application should set flags explicitly
  - listening socket remains unconnected and can accept next connection
  - connections are accepted in the order they were queued

Socket Functions

- Function `accept` (cont.):
  - blocks unless non-blocking flag is set for descriptor
  - listening socket indicates readiness for a new connection by readable event (f. `select`), even when select returns, such a connection may be already lost due to network errors, thus accept will block. To avoid this situation, non-blocking mode should be set on the socket.

Socket Functions

- Function `accept` and `getpeername` (cont.):
  - can obtain peer socket address
    - via output parameters
    - if address buffer is too small, the address will be **silently** truncated (the function cannot overflow the buffer)
    - if connection is made by unbound peer the value of the address is unspecified (udp and tcp use bounded clients only)
    - there is a very interesting discussion on the type of last argument to accept (see NOTES in Linux manual on the `accept` function)
    - On HP-UX does not support AF_UNIX

Socket Functions

- Functions `socket`, `socketpair`, `listen`, `accept` and `bind` (cont.):
  - from linux manual:

POSIX.1-2001 does not require the inclusion of <sys/types.h>, and this header file is not required on Linux. However, some historical (BSD) implementations required this header file, and portable applications are probably wise to include it.

Socket Functions

- Operations on socket
  - blocks if operation buffer is full (and flag O_NONBLOCK is not set)
  - success of operation does not mean the delivery, only successful sending
  - if datagram is received it must be read at once, otherwise the remaining part will be truncated !
  - can report errors from previous operations (pending network errors)
  - using `read/write` functions on connected socket

Socket Functions

- Operations on socket (cont.)
  - with `recv/send` functions on connected socket
    - works as `read/write`
    - process can apply socket specific flags (POSIX list)
      - MSG_OOB – out of band data
      - MSG_EOR – end of record marker (not supported on byte stream)
      - MSG_PEEK – do not remove data from the buffer
      - MSG_WAITALL – the read shall not be terminated in the middle of a message (but still can be interrupted before any data is actually read)
      - <u>flag MSG_DONTWAIT is Linux specific !</u>
      - <u>flag MSG_NOSIGNAL is Linux specific !</u>

Socket Functions

- Operations on socket (cont.)
  - with `recvfrom/sendto` functions
    - works as `recv/send`
    - receiving process can learn peer address
    - sending process can address the message (datagrams only)
  - with `recvmsg/sendmsg` functions
    - works as `recvfrom/sendto`
    - sends messages with control information

Socket Functions

- Closing a socket
  - with `close` function
  - with `shutdown` function a process can shut only a part of a connection
    - SHUT_RD – process stops reading from the socket any incoming data should be rejected, process can still send data. If process tries to read from such a socket it will get EOF marker. Peer may get EPIPE or data may be silently discarded, in practice data is still received and queued, can be read in some systems !
    - SHUT_WR - process stops writing to socket any outgoing data will be rejected, but process still can read from the socket. If process tries to write to such a socket it will be send SIGPIPE (or EPIPE error)
    - SHUT_RDWR – closes both ways, still requires f.`close` .

Unix Domain Sockets

- Communication between processes on the same computer
  - socket name is simply a name in local file system
  - processes must have write (w) and search (x) permissions to the directory where socket is created
  - process must have read and write permissions to the directory containing the socket it is connecting to
  - permissions on local sockets can be ignored by some systems and should not be used as security measures
  - usually local sockets are put in `/tmp`

Unix Domain Sockets

- both communication styles (*SOCK_STREAM*, *SOCK_DGRAM*) are supported in this namespace
- name in file system is necessary only to establish connection
- socket file persists and can/should be deleted by the process
- local sockets can not be used to connect remote process (even through nfs or afs) at the moment but it is possible that it will change in the future.

Unix Domain Sockets

- for better portability process should not depend on the fact that the other process resides on the same machine
  - pid numbers
  - byte order
  - sending structures
- for sake of simplicity students can assume that local sockets are truly local in laboratory applications
- Linux implementation does not support out of bound data transfer on local sockets

Unix Domain Sockets

- Address in unix domain sockets are stored in `struct sockaddr_un:`

  `sa_family_t sun_family`

  `char sun_path[108]`
- The length of this structure (passed to bind) should be calculated as:

  `sizeof(sa_family_t)+strlen(a.sun_path)`
- The `SUN_LEN` macro returns proper size
- <u>**Do not use `sizeof(sockaddr_un)`!!!**</u>

Internet Domain Sockets

- Communication between processes over the tcp/ip network
  - IPv4 vs. IPv6, at the moment only IPv4 is common and will be discussed here
  - socket name consists of IP address (x.x.x.x) and the port number **both represented in network byte order** (see byte order)
  - both communication styles (*SOCK_STREAM*, *SOCK_DGRAM*) are supported in this namespace
  - **no permission control built into AF_INET sockets !!!**

Internet Domain Sockets

- Address in internet domain sockets are stored in `struct sockaddr_in`:

```
sa_family_t sin_family
struct in_addr sin_addr
unsigned short int sin_port
```

- Where `in_addr` is a one-filed structure:

```
u_int32_t s_addr;
```

- The length of this structure (passed to bind) should be calculated as

```
sizeof(sockaddr_in)
```

Internet Domain Sockets

- Special addresses (**<u>host byte order</u>**):
  - `uint32_t INADDR_LOOPBACK` (127.0.0.1)
  - `uint32_t INADDR_ANY` (0.0.0.0)
    - very useful as server address
    - binds to all local interfaces
    - if socket is unbound prior to connect or send it will be automatically bound to INADDR_ANY and random free port
  - `uint32_t INADDR_BROADCAST` (255.255.255.255)
    - to send broadcast datagrams (SOCK_DGRAM)
  - `uint32_t INADDR_NONE` (-1==255.255.255.255)
    - error indicator

Internet Domain Sockets

- IP address by type:
  - unicast – addressing of single peer
    - only option for SOCK_STREAM
  - multicast – addressing a set of peers in the network
    - not available by default, popular, speeds up streaming
    - will not be discussed here
  - broadcast – addressing all the peers in the network
    - useful communication method in local networks
    - can be blocked by routers and local firewalls and we experienced many problems with broadcast tasks during labs

Internet Domain Sockets

- Socket address (cont.)
  - to convert IP string e.g. "194.29.178.1" to `uint32_t` POSIX defines `inet_addr` function
    - returns `INADDR_NONE` as error, it may be mistaken with `INADDR_BROADCAST`
    - `inet_addr` returns an address in **network byte order**
    - GNU extension `inet_aton` reports errors more reliably but is not standardized
  - to convert binary address in network byte order to string POSIX defines `inet_ntoa`
    - returns string in static buffer – not thread safe

Internet Domain Sockets

- Socket address (cont.)
  - as socket requires IP number to bind, domain name must be resolved (DNS) with `gethostbyname`
    - on success returns `hostent` structure, where address in question is available as field `char *h_addr_list[0]` (must be cast to `struct in_addr`)
    - <u>errors are reported in global variable **h_errno** not errno !!!</u>
    - can be used with domain names and IP numbers as strings on Linux (POSIX does not define it)
    - returns addresses in network byte order
    - returned address may be stored in static buffer
    - not thread safe

Internet Domain Sockets

- Datagrams (packets)
  - if size of datagram is larger than MTU on given socket sending will return with EMSGSIZE error
  - process can test MTU with `getsockopt` or experiment with the sizes and EMSGSIZE or assume small datagrams (less than 576)
  - `read/write/send/recv/sendto` and `recvfrom` will perform atomically on datagrams
  - if datagram is received it must be read at once, otherwise the remaining part will be truncated !

Internet Domain Sockets

- Datagrams (packets) cont.
  - if datagram is lost, application must retransmit it, usually if response is not received after some time-out packet can be considered lost
  - usually messages are sent in single packets
  - application logic must be ready for lost packets and duplications
  - on Linux datagrams are reliable, duplications and mixed order or delivery are not possible, but this is not POSIX behaviour
  - Linux also has a reliable datagram SOCK_RDM socket type

Internet Domain Sockets

- Binary stream
  - reliable
  - usually only one process operates on each end of connection
  - message boundaries are not preserved
  - **<u>there is no atomic message size !!!</u>**
  - **<u>read/write operation can be interrupted at any stage (EINTR)</u>**

Internet Domain Sockets

- Binary stream (cont.)
  - usually connection is bidirectional (read/write), process can modify file flags or use `shutdown` function to limit access
  - file position is fixed, reading from beginning, writing at the end (FIFO order)
  - read buffer is independent from write buffer

Sockets - Examples

- How to make local SOCK_STREAM socket :

```
int make_socket(char* name , struct sockaddr_un \
*addr) {
    int socketfd;
    if ((socketfd = socket(PF_UNIX,SOCK_STREAM, \
                          0)) < 0) ERR("socket");
    memset(addr, 0, sizeof(struct sockaddr_un));
    addr->sun_family = AF_UNIX;
    strncpy(addr->sun_path, name, sizeof( \
                              addr->sun_path)-1);
    return socketfd;
}
```

Sockets - Examples

- How to bind and start listening on local SOCK_STREAM socket :

```
#define BACKLOG 3

int bind_socket(char *name) {
    struct sockaddr_un addr;
    int socketfd;
    if (unlink(name) < 0 && errno != ENOENT)
          ERR("unlink");
    socketfd = make_socket(name,&addr);
    if (bind(socketfd, (struct sockaddr*) &addr, \
          SUN_LEN(&addr)) < 0) ERR("bind");
    if (listen(socketfd, BACKLOG) < 0) ERR("listen");
    return socketfd;
}
```

Sockets - Examples

- How to connect to local SOCK_STREAM socket :

```
int connect_socket(char *name) {
  struct sockaddr_un addr; int socketfd;
  socketfd = make_socket(name,&addr);
  if (connect(socketfd,(struct sockaddr*) &addr, \
                              SUN_LEN(&addr)) < 0) {
      if (errno != EINTR) ERR("connect");
      else {
         fd_set wfds ; int status;
         socklen_t size = sizeof(int);
         FD_ZERO(&wfds); FD_SET(socketfd, &wfds);
         if (TEMP_FAILURE_RETRY(select(socketfd+1, \
             NULL, &wfds, NULL, NULL)) < 0) ERR("select");
         if (getsockopt(socketfd, SOL_SOCKET, SO_ERROR, \
               &status,&size)<0) ERR("getsockopt");
         if(0!=status) ERR("connect");
      } }
  return socketfd; }
```

Sockets - Examples

- How to accept connection on SOCK_STREAM socket :

```
int add_new_client(int sfd,fd_set* base_rfds, \
                                       int* fdmax){
    int fd;
    if((fd=TEMP_FAILURE_RETRY(accept(sfd,NULL,\
        NULL)))<0)ERR("accept");
    FD_SET(fd, base_rfds);
    *fdmax=(*fdmax<fd?fd:*fdmax);
    return 1;
}
```

Sockets - Examples

- How to make internet server SOCK_DGRAM socket :

```
int make_socket(uint16_t port) {
    struct sockaddr_in name;
    int sock, t=1;
    sock = socket(PF_INET,SOCK_DGRAM,0);
    if (sock < 0) ERR("socket");
    name.sin_family = AF_INET;
    name.sin_port = htons(port);
    name.sin_addr.s_addr = htonl(INADDR_ANY);
    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
        &t, sizeof(t))) ERR("setsockopt");
    if (bind(sock,(struct sockaddr*) &name, \
        sizeof(name)) < 0) ERR("bind");
    return sock;
}
```

Sockets - Examples

- How to receive datagrams:

```
int16_t recv_datagram(int sock, struct sockaddr_in
*addr) {
    int16_t buf;
    socklen_t len = sizeof(struct sockaddr_in);
    if (TEMP_FAILURE_RETRY(recvfrom(sock, &buf, \
       sizeof(int16_t), 0, (struct sockaddr*) addr, \
&len)) \ < sizeof(int16_t))
       ERR("recvfrom");
    return ntohs(buf);
}
```

Sockets - Examples

- How to send datagrams:

```
int send_datagram(int sock, struct sockaddr_in addr,
int16_t msg) {
    int status;
    int16_t buf = htons(msg);
    status = TEMP_FAILURE_RETRY(sendto(sock, &buf, \
       sizeof(int16_t), 0, (struct sockaddr*) \
    &addr, sizeof(addr)));
    if (status < 0 && errno != EPIPE && errno ==
ECONNRESET)
        ERR("sendto");
    return status;
}
```

Sockets - Examples

- How to lookup domain name in DNS:

```
struct sockaddr_in make_address(char *address,
uint16_t port) {
    struct sockaddr_in addr;
    struct hostent *hostinfo;
    addr.sin_family = AF_INET;
    addr.sin_port = htons (port);
    hostinfo = gethostbyname(address);
    if (hostinfo == NULL) ERRH("gethost:");
    /*h_errno*/
    addr.sin_addr =
        *(struct in_addr*)hostinfo->h_addr;
    return addr;
}
```