Internet Sockets — an introduction

Dariusz Bismor

Silesian University of Technology
Institute of Automatic Control
Measurements and Control Systems Group

October 2021

Socket Basics

Our goal

Our Goal

To write an Internet server

Our goal

- We will start small
- We will start by writing a server that merely opens a port
- Later, we will add functionality to receive data
- We will be accessing this server with `telnet` (or `nc`) command
- We will also write a simple client
- Later, we will add functionality to answer requests
- Finally we will discover how little more we need to serve files

You may start programming right away (if you feel confident), or listen to the presentation

Addresses in IP version 4 protocol

- IP version 4 uses 32 bits (4 bytes) to store an address
- This limits the number of addresses available to 2^{32} (4 294 967 296)
- There are addresses reserved for special purposes: some 18 mln addresses are reserved for private networks, some 16 mln addresses are reserved for multicasting
- Private network addresses are: 10.0.0.0 – 10.255.255.255; 172.16.0.0 – 172.31.255.255 i 192.168.0.0 – 192.168.255.255
- Originally the IP addresses were divided into classes (A, B, C, D & E), this early design was replaced by CIDR mechanism (*Classless Inter-Domain Routing*), allowing for slowing the growth of address routing tables, and helping to prevent the rapid exhaustion of IPv4 addresses
- In spite of this, the IP version 4 address space is already exhausted
- The address space exhaustion can be mitigated by use of NAT (*Network Address Translation*) and use of private networks

Addresses in IP version 6 protocol

- IP version 6 uses 128 bits (16 bytes) to store an address
- This limits the number of addresses available to 2^{128} (about 3.44×10^{38})
- After migration to this version of IP protocol NAT will be no longer necessary
- The exhaustion of this address space cannot be expected in predictable future (5×10^{28} addresses for each human being, 4.5×10^{15} addresses for each star in observable Universe)
- (The exhaustion of IP version 4 address space has not been foreseen either)

Sockets in Unix systems

- Each data transfer between two Unix programs can be regarded to be data read/write of some special files
- Not only files on storage devices are considered as files in the previous statement, but also data streams, terminals, FIFO queues, external devices or network connections
- A socket is a bidirectional communication device that can be used for IPC (inter-process communication)
- There are different types of sockets: Internet Sockets, Unix Sockets, X.25 Sockets etc.
- Unix sockets are standardized by POSIX standard (full name: *POSIX Local Inter-process Communication Sockets*)
- Internet Sockets are bidirectional devices to be used for communication over Internet

Internet socket address

The structure for storing Internet socket address is defined as (from `<sys/socket.h>`):

```
struct sockaddr{
  unsigned short int   sa_family; /* AF_INET */
  char                 sa_data[14];
}
```

- This structure generalizes more types of sockets
- IP v.4 address requires 4 bytes, plus additional 2 bytes for port number, total 6 bytes
- The remaining 8 bytes should be nulled (zeroed)

IP socket types

- Stream Sockets (SOCK_STREAM) — sockets for reliable duplex connections (TCP)
- Datagram Sockets (SOCK_DGRAM) — sockets for unreliable half-duplex connections (UDP)
- Raw Sockets (SOCK_RAW) — sockets for connections with user-defined transport layer
- Sequenced Packet Sockets (SOCK_SEQPACKET) — sockets for sequential, reliable duplex connection
- RDM Sockets (SOCK_RDM) — sockets for reliable, half-duplex connection

Byte order

There are two ways to store bytes in computer word

- “older” byte first (MSB, *Most Significant Byte*), also known as Big-Endian Byte Order or Network Byte Order
- “younger” byte first (LSB, *Least Significant Byte*), also known as Little-Endian Byte Order or Host Byte Order

Each operating system defines the following functions for conversion of byte order (declarations in `<netinet/in.h>`:

htons() — „host to network” for **short** data

htonl() — „host to network” for **long** data

ntohs() — „network to host” for **short** data

ntohl() — „network to host” for **long** data

IP v.4 address

The structure for storing IP v.4 socket address is defined as (from `<netinet/in.h>`):

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

struct in_addr{
  unsigned int          s_addr;
}
```


IP v.4 address

Example:

```
const unsigned short int PORT = 1234;
struct sockaddr_in Sender;

Sender.sin_family = AF_INET;
Sender.sin_addr.s_addr = inet_addr("192.168.1.1");
Sender.sin_port = htons(PORT);
memset( &(Sender.sin_zero), '\0', 8 );
```

Disadvantage: no validation of address conversion!
Function `inet_addr()` returns INADDR_NONE (usually -1) in case of an error. Unfortunately, -1 is also valid IP address 255.255.255.255

(To use `inet_addr()` it is necessary to include `<arpa/inet.h>`. To use `memset()` it is necessary to include `<string.h>`)

IP v.4 address

Another example:

```
const unsigned short int PORT = 1234;
struct sockaddr_in Sender;

Sender.sin_family = AF_INET;
if( !inet_aton("192.168.1.1", &(Sender.sin_addr)) ){
  printf("Błąd konwersji adresu");
  exit(10);
};

Sender.sin_port = htons(PORT);
memset( &(Sender.sin_zero), '\0', 8 );
```

Disadvantage: less portability

Function `inet_aton()` (ascii-to-network) returns nonzero if the conversion succeeded, and zero when the address is invalid. Function is also declared in `<arpa/inet.h>`.

IP v.4 address

Reverse conversion:

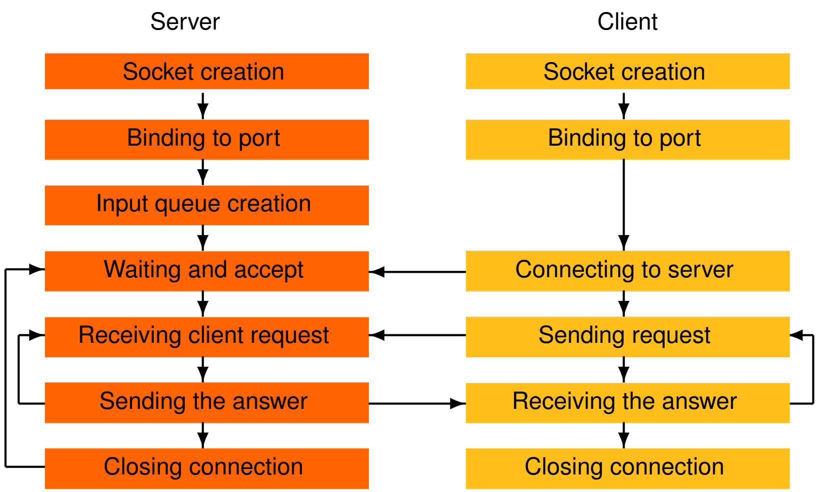
```
printf("Sender address is %s", inet_ntoa(Sender.sin_addr) );
```

Function `inet_ntoa()` (network-to-ascii):

- has prototype in `<arpa/inet.h>` header file
- accepts parameter of structure `in_addr` type, with IP address in NBO,
- returns a pointer to static buffer with address in dotted notion,
- static buffer means always the same buffer, next call to `inet_ntoa()` overwrites previous content.

TCP/IP Communication

Client-server communication



Socket creation

```
int socket(int domain, int type, int protocol)
```

- parameter *domain* defines domain: AF_UNIX, AF_INET, AF_INET6, AF_X25, etc.
- parameter *type* defines communication type: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, SOCK_SEQPACKET, SOCK_RDM
- parameter *protocol* defines protocol type, if there are few, most frequently it has 0 value
- on success function returns file descriptor for the new socket
- on error function returns -1 and sets global variable `errno`
- function is defined in `sys/socket.h` header file.

(A file descriptor in Unix systems is an abstract key for accessing a file, including special files, like directories, block devices, FIFOs or sockets. Technically, a file descriptor is an index for an entry in a kernel-resident data structure containing the details of all open files. In POSIX this data structure is called a file descriptor table, and each process has its own file descriptor table with at least three entries for *stdin, stdout* and *stderr*.)

Binding a socket to a port

```
int bind(int sockfd, struct sockaddr *MyAddress,
                                    socklen_t addrlen);
```

- parameter *sockfd* should be a socket returned by `socket()`
- parameter *MyAddress* should contain IP address and selected port
- parameter *addrlen* should contain address field length (its type is usually **unsigned int**)
- on error function returns -1 and sets `errno`
- if port number in *MyAddress* structure is set to 0, `bind()` will select a port from available ports (useful at client side)
- if the `sin_addr.s_addr` field of *MyAddress* structure is set to value `INADDR_ANY`, the socket is bound to every network interface

Listening on port

```
int listen(int sockfd, int backlog);
```

- parameter *sockfd* should be a socket returned by `socket()`
- parameter *backlog* defines waiting queue length (usually 5. . . 10)
- on error function returns -1 and sets `errno`

Connecting to a remote port

```
int connect(int sockfd, struct sockaddr *RemoteAddress,
                                    socklen_t addrlen);
```

- parameter *sockfd* should be a socket returned by `socket()`
- parameter *RemoteAddres* should contain the IP address of a server to connect to
- parameter *addrlen* should contain address field length
- on error function returns -1 and sets `errno`
- if there was no call `bind()` before `connect()`, `bind()` will be called automatically, with random selected port

Accepting a remote connection

```
int accept(int sockfd, struct sockaddr *Address,
                              socklen_t *addrlen);
```

- parameter *sockfd* should be a socket returned by `socket()`
- parameter *Address* should be a pointer to a structure where the remote connection data will be stored
- parameter *addrlen* should contain address field length
- on success function returns new connection file descriptor
- on error function returns -1 and sets `errno`
- TCP/IP API level does not allow to check who is trying to connect to a server before accepting the connection

Server code example

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
const unsigned short int PORT = 1234;

int main(void){
  int fd, connectionfd;
  struct sockaddr_in ownAddress;
  struct sockaddr_in remAddress;
  int addrlen;

  if( (fd = socket(PF_INET, SOCK_STREAM, 0)) == -1 ){
    perror( "socket" );
    exit(1);
  };
```


Server code example

```
  ownAddress.sin_family = AF_INET;
  ownAddress.sin_port = htons(PORT);
  ownAddress.sin_addr.s_addr = INADDR_ANY;
  memset(&(ownAddress.sin_zero), '\0', 8);

  if( bind(fd, (struct sockaddr *) &ownAddress,
                  sizeof(struct sockaddr)) == -1){
    perror( "bind" );
    exit(2);
  }

  if( listen(fd, 5) == -1){
    perror( "listen" );
    exit(3);
  }
```


Server code example

```
  addrlen = sizeof(struct sockaddr_in);
  if( (connectionfd = accept( fd,
        (struct sockaddr *)&remAddress, &addrlen)) == -1 ) {
    perror( "accept" );
    exit(4);
  }

  printf("Connection from: %s\n", inet_ntoa(remAddress.sin_addr) );

  close( connectionfd );
  close( fd );
  return 0;
}
```

Client code example

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
const unsigned short int PORT = 1234;
const char * SERVER = "192.168.1.1";

int main(void){
  int fd;
  struct sockaddr_in remoteAddress;

  if( (fd = socket(PF_INET, SOCK_STREAM, 0)) == -1 ){
    perror( "socket" );
    exit(1);
  };
```


Client code example

```
  remoteAddress.sin_family = AF_INET;
  remoteAddress.sin_port = htons(PORT);
  remoteAddress.sin_addr.s_addr = inet_addr( SERVER );
  memset(&(remoteAddress.sin_zero), '\0', 8);

  if( (connect(fd, (struct sockaddr *) &remoteAddress,
                sizeof(struct sockaddr)) == -1){
    perror( "connect" );
    exit(2);
  }

  printf("Success!\n");
  close( fd );
  return 0;
}
```


Receiving data from a socket

```
int recv(int sockfd, void *Buffer, size_t len, int flags);
```

- parameter *sockfd* parameter *sockfd* should be a socket returned by `socket()`
- parameter *Buffer* should contain a buffer to a memory location prepared to receive data
- parameter *len* should contain *Buffer* length
- parameter *flags* allows for change of default behavior
- on success function returns number of bytes received
- on error function returns -1 and sets `errno`
- function returns 0 in case when the remote computer closed the connection

Received data is placed in a text buffer!

Sending data through a socket

```
int send(int sockfd, const void *Data, size_t len, int flags);
```

- parameter *sockfd* should be a socket returned by `socket()`
- parameter *Data* should be a pointer to a memory location where the data to be send is stored
- parameter *len* should contain the number of bytes to send
- parameter *flags* allows for change of default behavior
- on success function returns number of bytes sent
- on error function returns -1 and sets `errno`
- the number of bytes sent may be smaller than the requested number of bytes to send!
- the only difference between `send()` and `write()` is the possibility to use the *flags*

Data transferred with `send()` are a sequence of bytes (chars)!

Sending a complete sequence

```
int sendall(int fd, const char *data, int *len){
  int transfered = 0;          // Bytes already sent
  int left = *len;             // Bytes still to send
  int i;
  while( transfered < *len ){
    i = send( fd, data+transfered, left, 0 );
    if( i == -1 ){ break; }; // Error!
    transfered += i;
    left -= i;
  };

  *len = transfered;       //Number of bytes successfully transfered
                           //will be returned with this parameter

  return i == -1 ? -1 : 0;
}
```


Server code example — contd.

```
  char *message = "Service temporary unavailable";
  int messlen;

  messlen = strlen(message);
  if( sendall(connectionfd, message, &messlen, 0) == -1 ){
    fprintf(stderr, "Only %d bytes was transferred "
                    "because of an error!\n", messlen);
  }else{
    printf("Transfer completed successfully!");
  }

  close( connectionfd );
  close( fd );
  return 0;
}
```


Closing a connection

```
int close(int sockfd); //from <unistd.h>
int shutdown(int sockfd, int mode); //from <sys/socket.h>
```

- parameter *sockfd* should be a socket returned by `socket()`
- parameter *mode* defines shutdown mode
- on success functions returns 0
- on error functions return -1 and set `errno`

Shutdown modes:

- SHUT_RD (=0) disallows further recvs
- SHUT_WR (=1) disallows further sends
- SHUT_RDWR (=2) disallows sends and recvs (same as `close()`)

Closing a connection

The right way to close a connection:

- finish sending data,
- call `shutdown()` with *mode* equal to SHUT_WR (finished sending on a socket)
- call `recv()` in a loop until it returns 0 (remote computer closed the connection)
- call `close()`

Unix daemons

Unix daemons

- A daemon (service) is a program running in a background with little or not user intervention
- Examples of daemons are:
  - Apache www server,
  - Postfix, Sendmail, Qmail mail servers,
  - named DNS server
- Daemons are intended to perform one and only one task
- Creating a daemon requires following a set of rules

Unix daemons

- A daemon should not communicate with user directly
- Daemons usually use a configuration file on startup
- Daemons usually use log files (or syslog) to output messages
- User interaction with daemons is usually performed through interface of some kind (GUI, socket, IOCTL, etc.)

Unix daemons

Basic structure

- Fork off the parent process
- Change file mode mask (umask)
- Open any logs for writing
- Create a unique Session ID (SID)
- Change the current working directory to a safe place
- Close standard file descriptors
- Enter actual daemon code

Forking

```
#include <unistd.h>

pid_t id;

id = fork();
if( id < 0 ){
  //Failure to create the child process
  perror( errno );
  exit(1);
}

if( id > 0 ){
  //We are in the parent, just exit
  exit(0);
}
// There goes the child code
```


Changing the file mode mask

```
#include <sys/types.h>
#include <sys/stat.h>

//umask( mode_t mask );

umask( 0 );
```

- Changing the mask for newly created files allows for further access to those files
- The operation makes the daemon independent from the mask used by the user executing the daemon
- Mask equal 0 means full read and write access

Openinig logs

- Opening logs is optional, but very useful
- Logs allow for debugging of the daemon code
- The daemon may open its own logs, or use syslog facility

```
#include <syslog.h>

openlog( "mydaemon", LOG_PID, LOG_DAEMON );
```

- The first parameter is the name to be prepended to each entry (typically daemon name)
- The second is the options (here: include PID)
- The last is the log facility (like kernel, user, mail, news, etc)

Writing through syslog

```
syslog( LOG_INFO, "%s", "This is your daemon starting!\n" );
```

- The first parameter controls the message level:
  - LOG_EMERG — system is unusable
  - LOG_ALERT — action must be taken immediately
  - LOG_CRIT — critical conditions
  - LOG_ERR — error conditions
  - LOG_WARNING — warning conditions
  - LOG_NOTICE — normal, but significant, condition
  - LOG_INFO — informational message
  - LOG_DEBUG — debug-level message, lowest priority

Creating a unique Session ID

- A child process must obtain its own process ID (PID)
- Otherwise, a child without a parent is considered an orphan
- Orphans are removed from the system after some time

```
pid_t sid;

sid = setsid();
   if( sid < 0 ){
     //Failure to obtain PID
     syslog( LOG_ERR, "Failed to create my own PID, "
         "setsid returned %d\n", sid );
     exit(1);
   }
```


Changing the current directory

- A daemon should change itc pwd to a location that is guaranteed to exist
- A safe choice here is the root folder ("/")

```
if((chdir("/")) < 0) {
  syslog( LOG_ERR, "Failed to change directory to /\n");
  exit(1);
}
```


Closing standard file descriptors

- A daemon cannot use a terminal, therefore it does not need standard file descriptors (stdin, stdout, stderr)
- Standard file descriptors should be closed because they may influence the security

```
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);
```


The last stage

Finally, the daemon enters the main loop (usually infinite) and is ready to serve the clients

Responding to signals

- Daemons are detached from the terminal, so how to close a daemon?
- Unix solution to this question is called “signals”
- Signals can be handled by processes asynchronically
- Signals SIGKILL and SIGSTOP cannot be handled or ignored
- For the list of signals user “man 7 signal”

Writing a signal handler

```
void signal_handler(int sig) {
    switch(sig) {
        case SIGHUP:
            syslog(LOG_INFO, "Received SIGHUP signal.\n");
            break;
        case SIGTERM:
            syslog(LOG_INFO, "Received SIGTERM signal, exiting!\n");
            finalize_and_exit();
            break;
      /* and so on */
        default:
            syslog(LOG_WARNING, "Unhandled signal");
            break;
    }
}
```


Catching a signal

```
struct sigaction sa;

sa.sa_handler = &signal_handler;
sigemptyset( &sa.sa_mask );
sa.sa_flags = SA_RESTART;

sigaction(SIGHUP, &sa, NULL );
sigaction(SIGTERM, &sa, NULL );
/* and so on */
```


Writing a web server

Forking

```
pid_t fork( void );
```

- The call to fork creates a child process that differs from the parent only with PID and PPID numbers
- This means that, to some degree, both parent and child have copies of the same parameters (the child does not inherit file blocks and signals)
- The value returned by `fork()`:
  - is equal to zero in the child process,
  - is equal to the PID of the child in the parent process
- Usually, in programs using `fork()`, there is an **if** condition with different code for the parent and a child

Server code example

```
/* All #include ommitted */
const unsigned short int PORT = 1234;
const int QLEN = 10;

void sigchld_handler(int s){
  while( wait(NULL) > 0 );
}

int main( void ){
  int fd, connectionfd;
  struct sockaddr_in myAddress, clientAddress;
  int siz;
  struct sigaction sa;
  int num, yes=1;
  char buffer[5];

/* Create a socket, bind to a port, call listen() */
```


Server code example

```
sa.sa_handler = sigchld_handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;

if( sigaction(SIGCHLD, &sa, NULL) == -1 ){
  perror("sigaction");
  exit(5);
}

while(1) {
  siz = sizeof(struct sockaddr_in);
  if( (connectionfd = accept( fd,
            (struct sockaddr *)&clientAddress, &siz)) == -1 ){
    perror("accept");
    continue;
  }
  printf( "Connection from: %s\n", inet_ntoa(clientAddress.sin_addr) );
```


Server code example

```
  if( !fork() ){
  /* Child process code here */
    close( fd ); /* Not needed in child */
    if( recv(connectionfd, buffer, sizeof(buffer), 0) == -1){
      perror("receive");
      close( connectionfd );
      exit(6);
    }

    /* Received command from client, compose the answer */

    if( send( connectionfd, buffer, strlen(buffer), 0) == -1){
      perror("send");
      close( connectionfd );
      exit(7);
    }
    close( connectionfd );
    exit(0); /* Child process gracefully closed */
```

Server code example

```
    }else{  /* To if( fork ) */
      close( connectionfd );
      /* Parent process does not need connected socket, */
      /* just go back to listening */
    }
  return 0;
}
```