Politechnika Łódzka
Wydział Elektrotechniki, Elektroniki, Informatyki i Automatyki

Network Programming

Lecture 3
TCP Client

Instytut Informatyki Stosowanej
Politechniki Łódzkiej

Dr Radosław Wajman

Socket address structure

- The structure contains the protocol specific addressing information that is passed from the user process to the kernel and vice versa

- Each of the protocols supported by a socket implementation have their own socket address structure

  sockaddr_*suffix*

  Where *suffix* represents the protocol family

  Ex: sockaddr_in – Internet/IPv4 socket address structure

  sockaddr_ipx – IPX socket address structure

Socket address structure (IPv4)

The generic socket address structure

```
typedef struct sockaddr {
   u_short sa_family;
   CHAR sa_data[14];
} SOCKADDR;
```

The internet/IPv4 socked address structure

```
typedef struct sockaddr_in {
   short sin_family;
    unsigned short sin_port;
   IN_ADDR sin_addr;
   CHAR sin_zero[8];
} SOCKADDR_IN, *PSOCKADDR_IN;
```

```
typedef struct in_addr {
    union {
        struct {
            UCHAR s_b1, s_b2, s_b3, s_b4;
             } S_un_b;
        struct {
            USHORT s_w1,s_w2;
             } S_un_w;
        ULONG S_addr;
       } S_un;
} IN_ADDR, *PIN_ADDR, *LPIN_ADDR;
```

```
sockaddr_in service;
service.sin_family = AF_INET;
service.sin_addr.s_addr =
                    inet_addr("127.0.0.1”);
service.sin_port = htons(3370);
```

IP address <->Domain names

```
sockaddr_in service;
service.sin_family = AF_INET;
service.sin_port = htons(3370);
service.sin_addr.s_addr = ?????;
```

IP address in TXT format:

```
.s_addr =
  inet_addr("127.0.0.1”);
```

How to assign the TXT domain name address?

google.pl = 74.125.77.147 ?

```
#define AF_UNSPEC     0        // unspecified
#define AF_UNIX       1        // local to host (pipes, portals)
#define AF_INET       2        // internetwork: UDP, TCP, etc.
#define AF_IMPLINK    3        // arpanet imp addresses
#define AF_PUP        4        // pup protocols: e.g. BSP
#define AF_CHAOS      5        // mit CHAOS protocols
#define AF_NS         6        // XEROX NS protocols
#define AF_IPX        AF_NS    // IPX protocols: IPX, SPX, etc.
#define AF_ISO        7        // ISO protocols
#define AF_OSI        AF_ISO   // OSI is ISO
#define AF_ECMA       8        // european computer manufacturers
#define AF_DATAKIT    9        // datakit protocols
#define AF_CCITT      10       // CCITT protocols, X.25 etc
#define AF_SNA        11       // IBM SNA
#define AF_DECnet     12       // DECnet
#define AF_DLI        13       // Direct data link interface
#define AF_LAT        14       // LAT
#define AF_HYLINK     15       // NSC Hyperchannel
#define AF_APPLETALK  16       // AppleTalk
#define AF_NETBIOS    17       // NetBios-style addresses
#define AF_VOICEVIEW  18       // VoiceView
#define AF_FIREFOX    19       // Protocols from Firefox
#define AF_UNKNOWN1   20       // Somebody is using this!
#define AF_BAN        21       // Banyan
#define AF_ATM        22       // Native ATM Services
#define AF_INET6      23       // Internetwork Version 6
#define AF_CLUSTER    24       // Microsoft Wolfpack
#define AF_12844      25       // IEEE 1284.4 WG AF
#define AF_IRDA       26       // IrDA
#define AF_NETDES     28       // Network Designers OSI & gateway
```

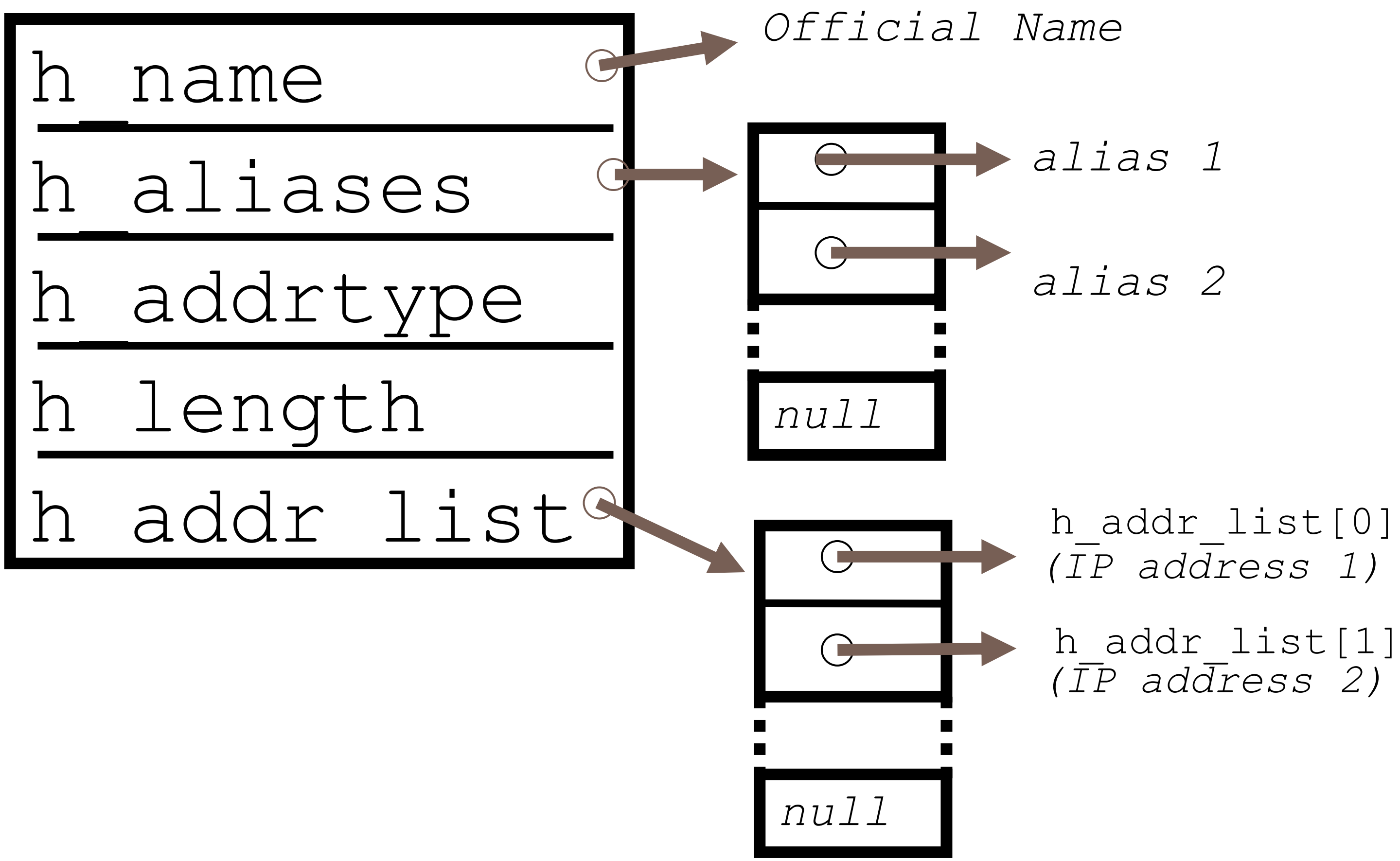
Queries for DNS

In order to send a query to DNS server and
to get know the IP address of the host having its domain name
the following function must be used:

```
hostent* h = gethostbyname("google.pl”);
if (h == NULL)
{
      printf(„error”); exit(1);
}
service.sin_addr = *(struct in_addr*)h->h_addr_list[0];
```

- The HOSTENT structure contains the address information,
- In the program it is not allowed to modify or deallocate it as well as any of its field.
- ONLY one copy of it is assigned to the thread,
    - Each call of **gethostbyname** or **gethostbyaddr** overwrites existing data ,

HOSTENT structure



All the IP addresses returned via the HOSTENT are in network byte order

Error handling: gethostbyname

- On error `gethostbyname` return null.

- `gethostbyname` sets the global variable `h_errno` to indicate the exact error:
  - `HOST_NOT_FOUND`
  - `TRY_AGAIN`
  - `NO_RECOVERY`
  - `NO_DATA`
  - `NO_ADDRESS`

Name/Address Conversion

- Protocol dependent DNS library functions
  - `gethostbyname`
  - `gethostbyaddr`

- Posix protocol *independent* functions
  - `getaddrinfo()`
    provides protocol-independent translation from an ANSI host name to an address
  - `getnameinfo()`
    provides protocol-independent name resolution from an address to an ANSI host name and from a port number to the ANSI service name
  - these functions were designed to support writing code that can run on many protocols (IPv4, IPv6)

POSIX: getaddrinfo()

```
int getaddrinfo(
     const char *hostname,
     const char *service,
     const struct addrinfo* hints,
     struct addrinfo **result);
```

- `hostname` is a hostname or an address string (dotted decimal string for IP).
- `service` is a service name or a decimal port number string.
- `getaddrinfo()`
  - *provides the combined functionality of* `gethostbyname()` *and* `getservbyname()`

addrinfo structure

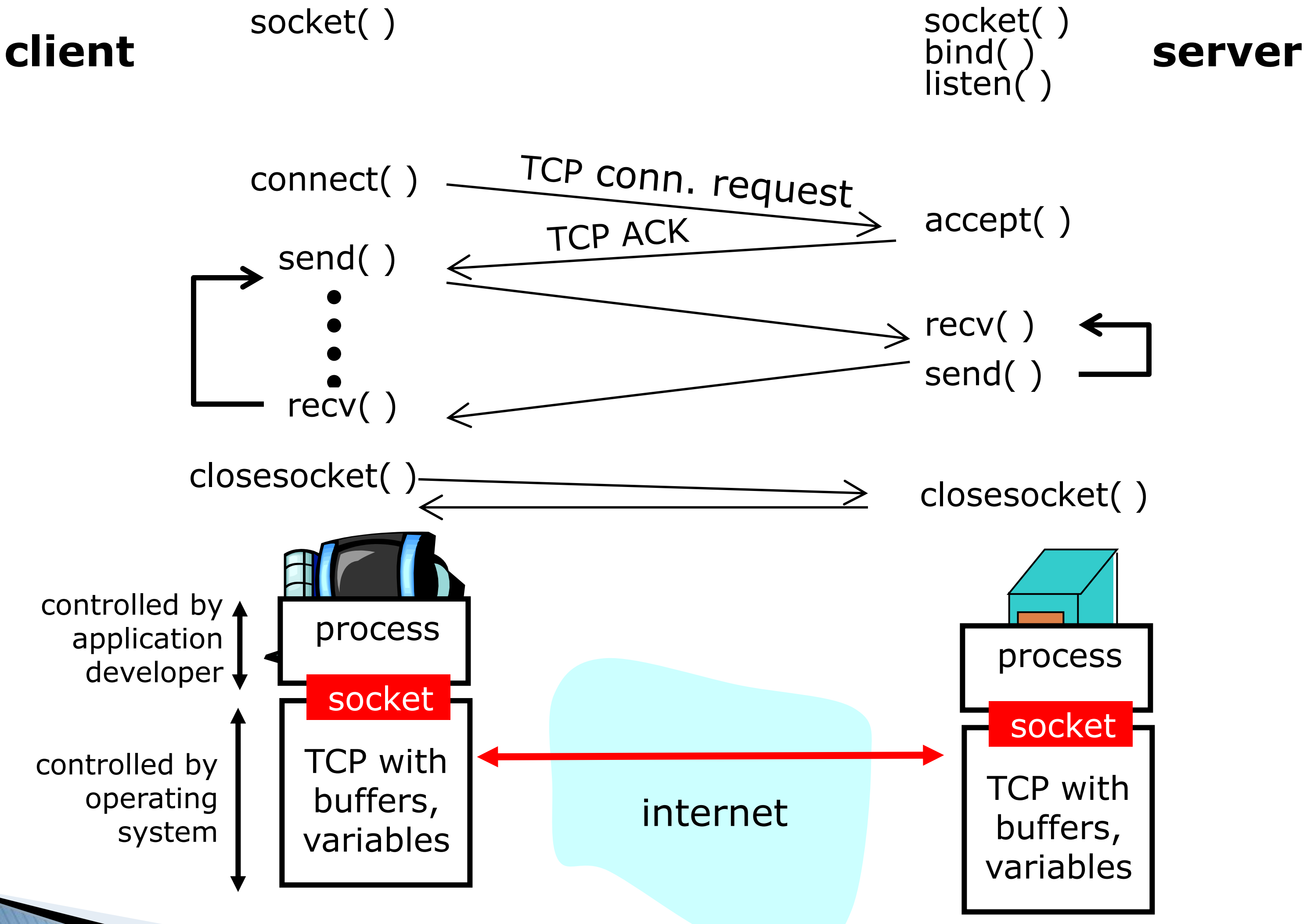
- `result` is returned with the address of a pointer to an `addrinfo` structure that is the head of a linked list.

```
struct addrinfo {
  int        ai_flags;
  int        ai_family;
  int        ai_socktype;
  int        ai_protocol;
  size_t          ai_addrlen;
  char            *canonname;
  struct   sockaddr *ai_addr;
  struct   addrinfo *ai_next;
};
```

used by socket()

used by:
bind()
connect()
sendto()

Client+server: connection-oriented



Client: Init socket [1]

socket ► connect ► send ► recv ► closesocket

```
int socket(int family, int type, int proto);
```

Creates in the system a new socket and assignes it to the protocol.

- `family` – protocols family:
  - AF_INET – for IPv4,
  - **AF_INET6** – for IPv6
- `type` – socket type:
  - SOCK_STREAM – stream socket (TCP),
  - **SOCK_DGRAM** – datagram socket (UDP),
  - **SOCK_RAW** – raw socket,
- `proto` – protocol (for type=**SOCK_RAW**):
  - 0 – default protocol (SOCK_STREAM=**TCP**, SOCK_DGRAM=**UDP**),
- **Result**: socket handle, or:
  - **INVALID_SOCKET**, error code from *WSAGetLastError* (Windows),
  - **–1**, error code from *errno* (Unix)

Client: Init socket [2]

socket ► connect ► send ► recv ► closesocket

**In case of Windows OS,
before using socket functions,
the WinSock library must be initialized.**

```
WSAData wsaData;
int nCode;
char errdesc[100];

if ((nCode = WSAStartup(MAKEWORD(1, 1), &wsaData)) != 0)
{
    sprintf(errdesc,
          „Error while initializing the WinSock library. Error %d", nCode);
    exit(1);
}
printf("WinSock: %s [%s]\n", wsaData.szDescription, wsaData.szSystemStatus);
printf("MaxSockets: %d\n", wsaData.iMaxSockets);
```

Client: Init socket [3]

socket ► connect ► send ► recv ► closesocket

Examples of ussage:

```
SOCKET sock_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

SOCKET sock_fd = socket(AF_INET, SOCK_STREAM, 0);

int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

SOCKET sock_fd = socket(AF_INET, SOCK_DGRAM, 0);
```

Client: Init connection

socket ► **connect** ► send ► recv ► closesocket

```
int connect(int sock, sockaddr *rmt, int rmtlen);
```

Initializes the connection with the remote host given in *rmt*. Processes the so called **active open**.

- *sock* – socket handle (returned by **socket**)
- *rmt* – pointer to the socket structure **sockaddr** with the remote host address, protocol and port number,
- *rmtlen* – length, in bytes, of the **sockaddr** structure for given protocol.
- **Result**: 0, or:
  - SOCKET_ERROR, error code from *WSAGetLastError* (Windows),
  - –1, error code from *errno* (Unix)

- **Blocking**: **connect** tries to connect with the remote host within a specified time.
- **Nonblocking**: Success or when the time is excided returns SOCKET_ERROR and error code = SAEWOULDBLOCK/EWOULDBLOCK.

```
connect(sock, (sockaddr*)&service, sizeof(sockaddr_in));
```

Client: Sending data to socket

socket ► connect ► **send** ► recv ► closesocket

Sending data!

```
int write(int sock, const char *buf, int buflen );
```

or

```
int send(int sock, const char* buf, int len, int flags);
```

Writes data in the socket sending buffer

- *sock* - socket handle (returned by **socket** or **accept**),
- *buf* - pointer to the buffer containing data to send,
- *buflen* - count of bytes to send,
- *flags* - flags, default 0,

- **Result**: count of actually sent bytes or:
  - SOCKET_ERROR, error code from *WSAGetLastError* (Windows),
  - -1, error code from *errno* (Unix)

- **Blocking**: **send** is waiting until the sending buffer picks up *buflen* bytes
- **Nonblocking**: **send** writes the data into the buffer (no less then 1) and returns count of actually written bytes. In case of lack of buffer space **send** returns SOCKET_ERROR and the error code = WSAEWOULDBLOCK/EWOULDBLOCK.

Client: Reading data from socket

socket ► connect ► send ► **recv** ► closesocket

Receiving data!

```
int read(int sock, char *buf, int buflen);
```

or

```
int recv(int sock, char *buf, int buflen, int flags);
```

Reads data from socket receiving buffer

- *sock* – socket handle (returned by **socket** or **accept**),
- *buf* – pointer to the buffer containing data to send,
- *buflen* – count of bytes to send,
- *flags* – flags, default 0,

- **Result**: 1 ≤ count of actually read bytes ≤ *buflen* , or:
  - 0 – when the connection was corrupted or remotely closed,
  - SOCKET_ERROR, error code from *WSAGetLastError* (Windows),
  - -1, error code from *errno* (Unix)

- **Blocking**: **recv** waits until the minimum (default 1) bytes will arrive to the buffer
- **Nonblocking**: **recv** reades as much data as arrived (no less then 1) and returns the count of actually read bytes.
  When the buffer is empty longer then the set TIMEOUT **recv** returnes SOCKET_ERROR and the error code = WSAEWOULDBLOCK/EWOULDBLOCK.

Client: Closing connection

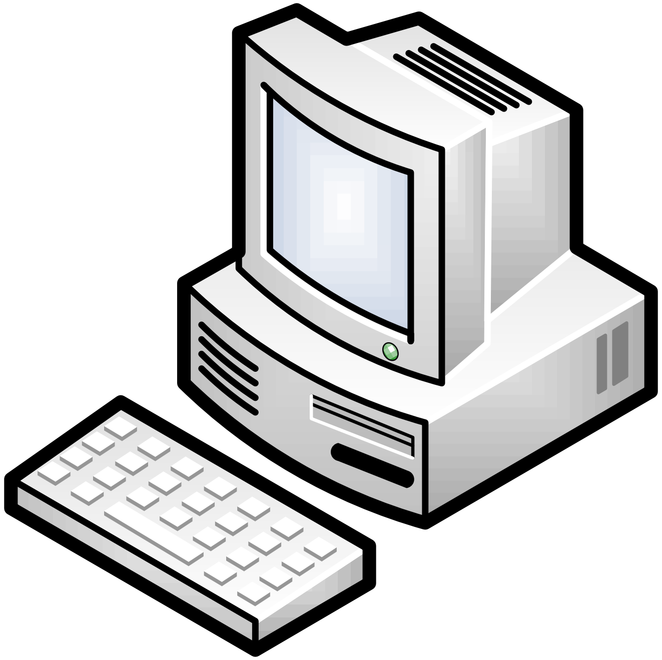
socket ► connect ► send ► recv ► **closesocket**

or
```
int closesocket(int sock);                    // windows
int close(int sock);                          // unix
```

Closes connection togheter with the socket.

Each active operation assigned to this socket will by cancelled.

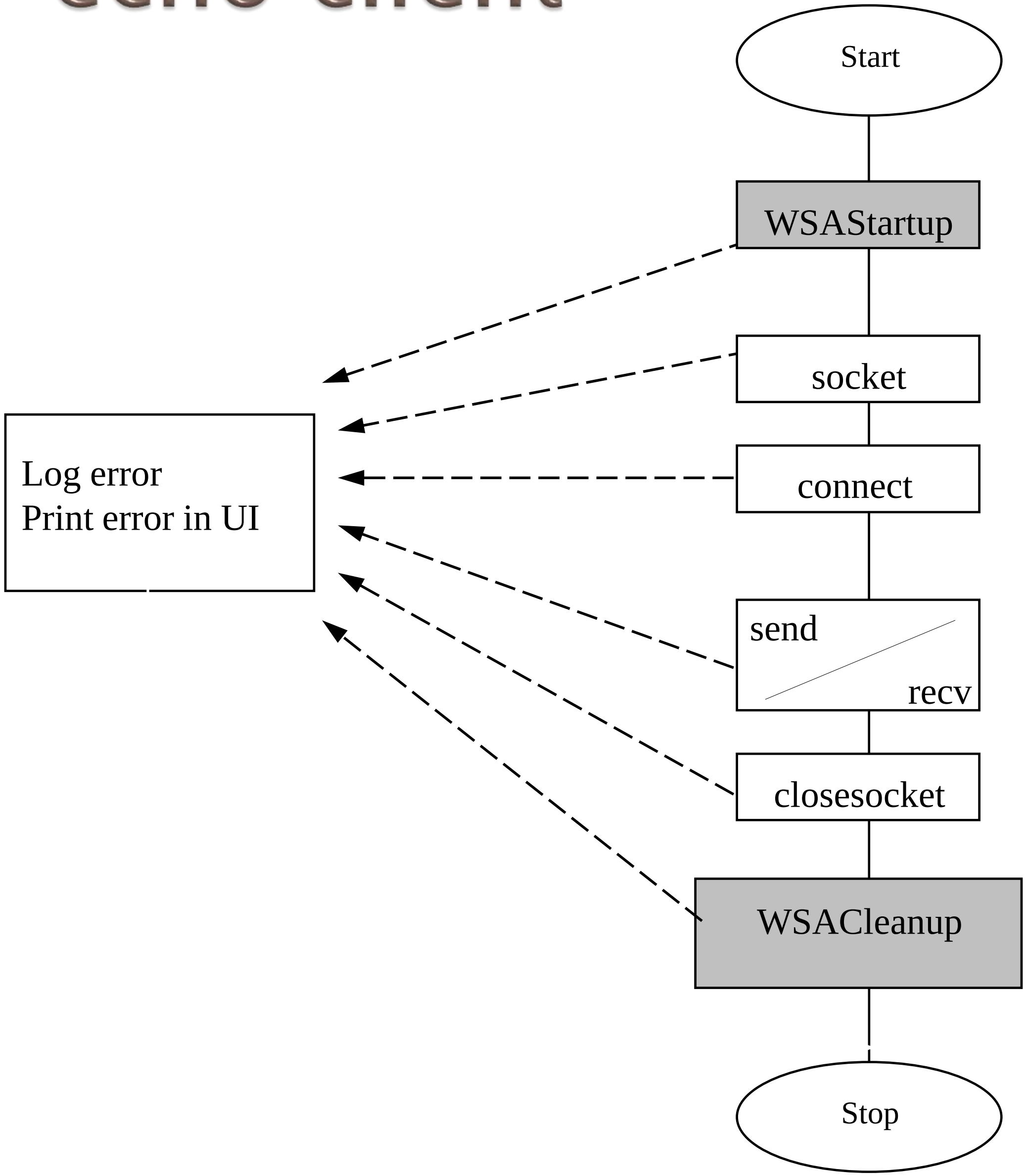
- *sock* - socket handle (returned by **socket** or **accept**),
- **Result**: 0 if socket was closed, or:
  - SOCKET_ERROR, error code from *WSAGetLastError* (Windows),
  - -1, error code from *errno* (Unix)



TCP

Client

Algoritm – echo client (C++)



TCP client: C++ implementation

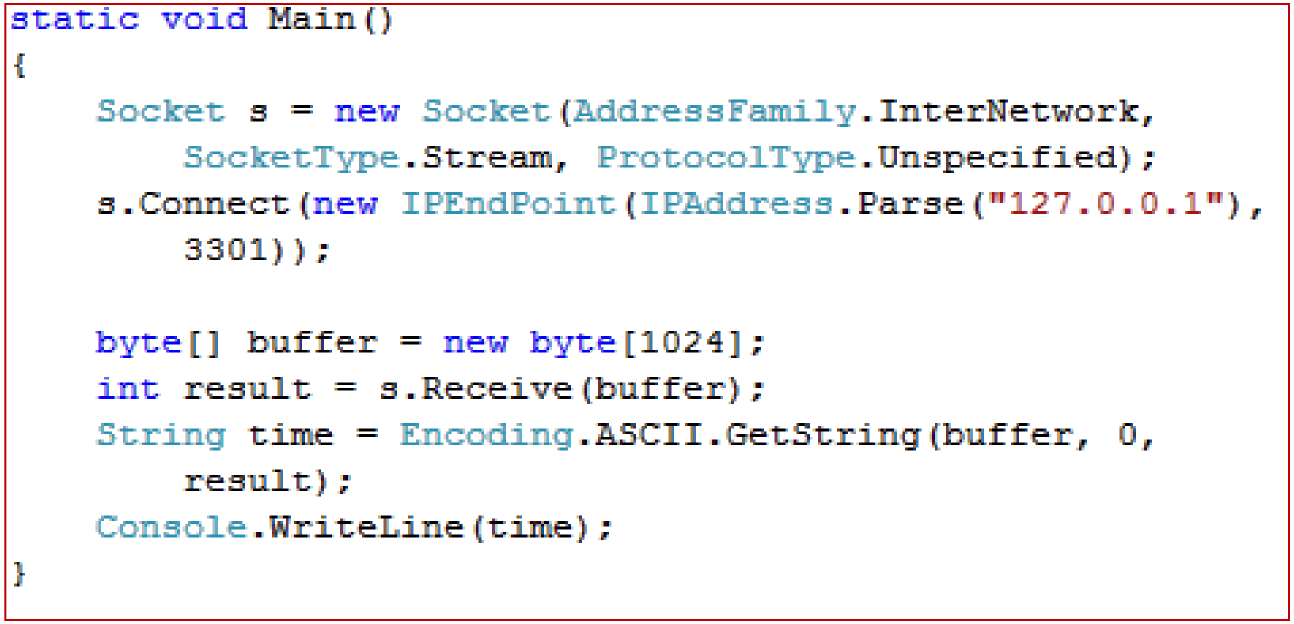
```
int main(int argc, char* argv[])
{
    WSAData data;
    int result;

    result = WSAStartup(MAKEWORD(2, 0), &data);
    if(result == 0){/*handle error*/};

    SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
    if(sock != INVALID_SOCKET){/*handle error*/};

    sockaddr_in service;
    service.sin_family = AF_INET;
    service.sin_port = htons(3301);
    service.sin_addr.s_addr = inet_addr("127.0.0.1");
    result = connect(sock, (sockaddr*)&service,
                     sizeof(sockaddr_in));
    if(sock != INVALID_SOCKET){/*handle error*/};

    char str[100];
    for(int i = 0; i < 3; i++) {
        if (!read_line(sock, str))
            break;
        printf("%d: %s", i, str);
    }
    closesocket(sock);
}
```

```
bool read_line(SOCKET sock, char* line)
{
    while(true)
    {
        int result = recv(sock, line, 1, 0);
        if (result == 0 || result ==
                               SOCKET_ERROR)
            return false;
        if (*line++ == '\n')
            break;
    }
    *line = '\x0';
    return true;
}
```

The server protocol/answer

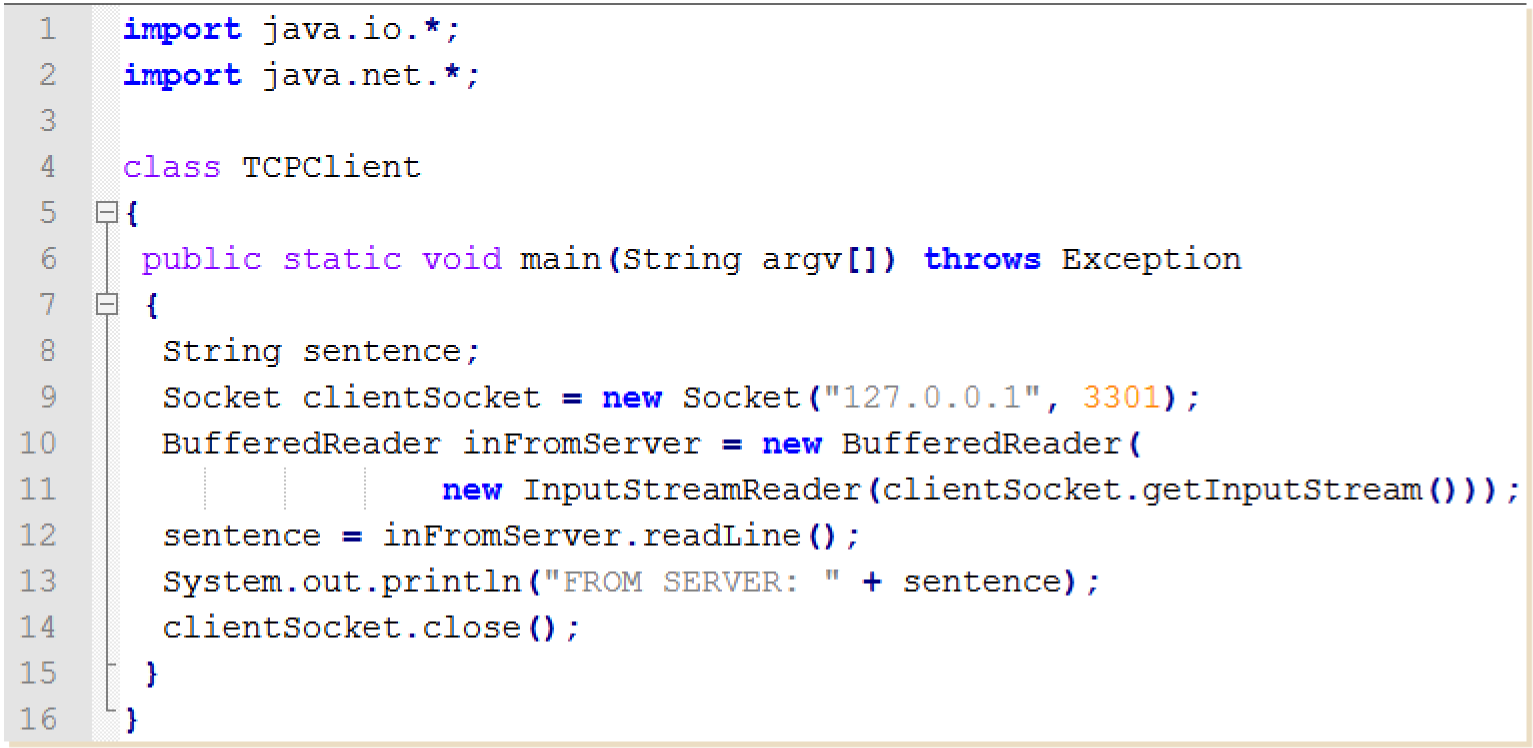
```
Data 11/10/2010\r\n
Godzina 17:53:41\r\n
Jestes klientem #1\r\n
```

TCP client: C# implementation



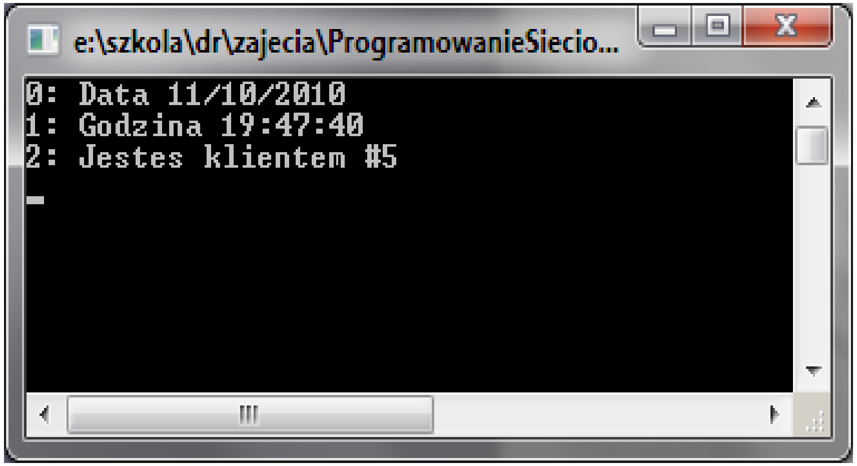
```
static void Main()
{
    Socket s = new Socket(AddressFamily.InterNetwork,
        SocketType.Stream, ProtocolType.Unspecified);
    s.Connect(new IPEndPoint(IPAddress.Parse("127.0.0.1"),
        3301));

    byte[] buffer = new byte[1024];
    int result = s.Receive(buffer);
    String time = Encoding.ASCII.GetString(buffer, 0,
        result);
    Console.WriteLine(time);
}
```

TCP client: JAVA implementation

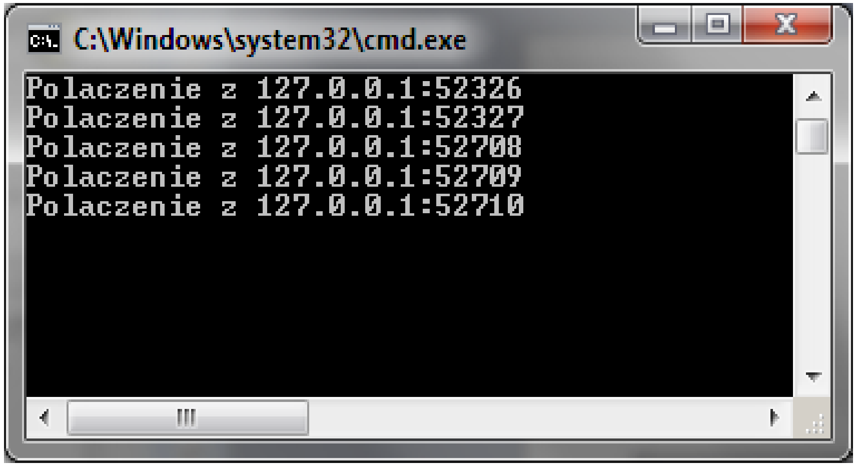


```java
import java.io.*;
import java.net.*;

class TCPClient
{
 public static void main(String argv[]) throws Exception
 {
  String sentence;
  Socket clientSocket = new Socket("127.0.0.1", 3301);
  BufferedReader inFromServer = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
  sentence = inFromServer.readLine();
  System.out.println("FROM SERVER:  " + sentence);
  clientSocket.close();
 }
}
```

TCP client and server: Tests



```
0: Data 11/10/2010
1: Godzina 19:47:40
2: Jestes klientem #5
_
```



```
Polaczenie z 127.0.0.1:52326
Polaczenie z 127.0.0.1:52327
Polaczenie z 127.0.0.1:52708
Polaczenie z 127.0.0.1:52709
Polaczenie z 127.0.0.1:52710
```

Server protocol / answer

```
Data 11/10/2010\r\n
Godzina 17:53:41\r\n
Jestes klientem #1\r\n
```

Error Handling

Issues and Ideas

System Errors

- In general, systems calls return a negative number to indicate an error.
  - We often want to find out what error.
  - Servers generally add this information to a log.
  - Clients generally provide some information to the user.

extern int errno;

- Whenever an error occurs, call *WSAGetLastError* to get error code
  - You can check error code for specific errors.
  - You can use support functions to print out or log an ASCII text error message.
- **`error code`** is valid only after a system call has returned an error.
  - System calls don't clear **`error code`** on success.
  - If you make another system call you may lose the previous value of **`error code`**.

Error codes

Error codes are defined in errno.h

```
EAGAIN   EBADF      EACCESS
EBUSY         EINTR      EINVAL
EIO      ENODEV     EPIPE
…
```

Support routines

- `void perror(const char *string);`
  - `stdio.h`
- `char *strerror(int errnum);`
  - `string.h`

General Strategies

- Include code to check for errors after every system call.
- Develop "wrapper functions" that do the checking for you.
- Develop layers of functions, each hides some of the error-handling details.

Example wrapper

```
int Socket( int f, int t, int p) {
  int n;
  if ( (n=socket(f,t,p)) < 0 ) ) {
    perror("Fatal Error");
    exit(1);
  }
  return(n);
}
```

What is fatal?

- How do you know what should be a fatal error (program exits)?
  - Common sense.
  - If the program can continue – it should.

but

  - if a server can't create a socket,
    or can't bind to it's port
    - there is no sense continuing…

Wrappers are great!

- Wrappers like those used in the text can make code much more readable.

Another approach

- Instead of simple wrapper functions, you might develop a *layered system*.
- The idea is to "hide" the **sockaddr** and error handling details behind a few custom functions or classes:

```
int tcp_client(char *server, int port);
int tcp_server(int port);
```

Layers and Code Re-use

- Developing general functions that might be re-used in other programs is obviously "a good thing".

- Layering is beneficial even if the code is not intended to be re-used:
  - hide error-handling from "high-level" code.
  - hide other details.
  - often makes debugging easier.

The best approach to handling errors?

- There is no *best approach*.
- Do what works for you.
- Make sure you check *all* system calls for errors!
  - Not checking can lead to security problems!
  - Not checking can lead to bad grades on lab projects!