DVGBO2

Introduction to Berkley Sockets

Based on the slides from Michael Welzl, Pål Halvorsen, Carsten Griwodz, Nikhil Shetty & Olav Lysne

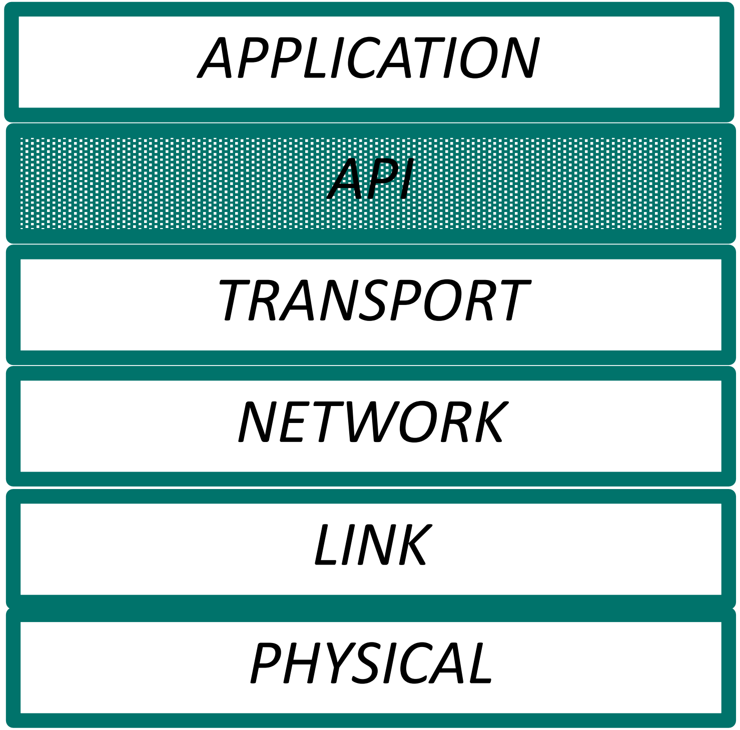
Goal

- Background
- Introduce socket API
- We will write two programs
  - A “client” and a “server”
- They will work as follows
  - The client sends the text “Hello world!” to the server
  - The server writes the received text on the screen
  - The server sends the received text back to the client and quits
  - The client writes the received text onto the screen and quits

What is an API?

- API stands for Application Programming Interface
- Interface to what? – in our case, it is an interface to use the network
- A connection to the transport layer
- Why do we need it?
  - One word – Layering
  - Functions at transport layer and below are very complex
  - E.g. Imagine having to worry about errors on the wireless link and signals to be sent on the radio.
  - Helps in code reuse.

API



APPLICATION

API

TRANSPORT

NETWORK

LINK

PHYSICAL

What is socket then?

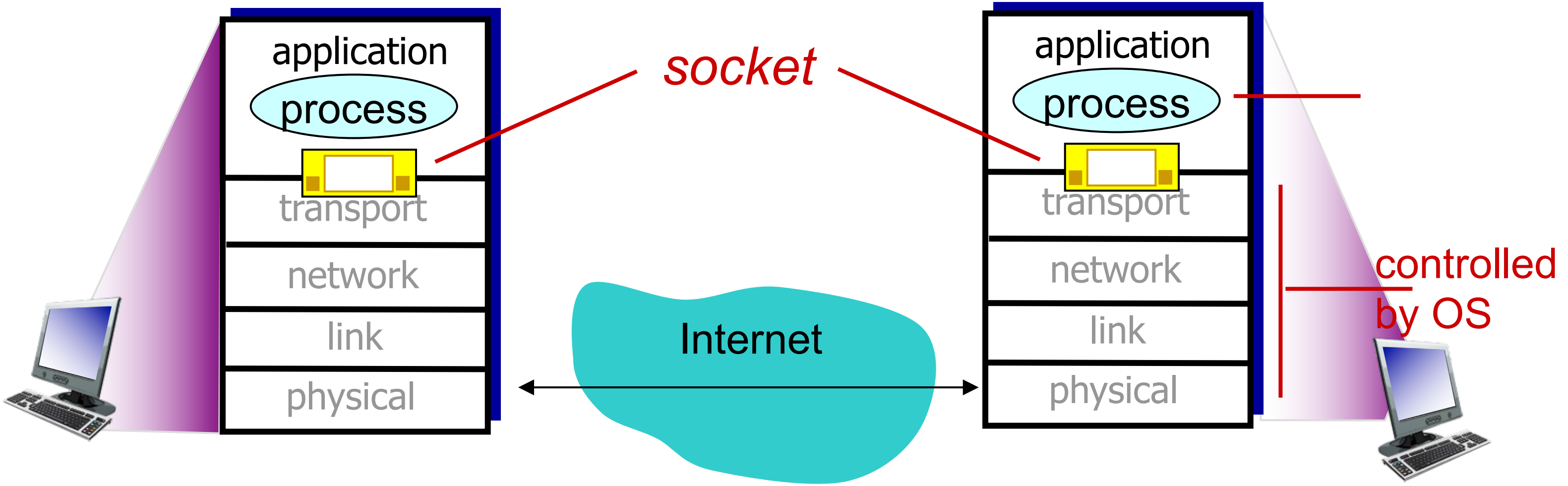
It is an abstraction that is provided to an application programmer to send or receive data to another process.

Data can be sent to or received from another process running on the same machine or a different machine.

In short, it is an end point of a data connection.

Sockets

- Application process send messages to Transport layer via sockets
- Application process is controlled by the developer
- Transport layer (TCP, UDP) is controlled by the OS



Sockets

- Identified by IP Address and Port number
- Berkeley Sockets in C
  - Released in 1983
  - Similar implementations in other languages

Sockets

Primitives	Meaning
SOCKET	Create a new communication endpoint.
BIND	Attach a local address to a SOCKET.
LISTEN	Shows the willingness to accept connections.
ACCEPT	Block the caller until a connection attempts arrives.
CONNECT	Actively attempt to establish a connection.
SEND	Send some data over connection.
RECEIVE	Receive some data from the connection.
CLOSE	Release the connection.

Ports

- Sending process must identify the receiver
  - Address of the receiving end host
  - Plus identifier (port) that specifies the receiving process
- Receiving host
  - Destination address uniquely identifies the host
- Receiving process
  - Host may be running many different processes
- Destination port uniquely identifies the socket
  - Port number is a 16-bit quantity

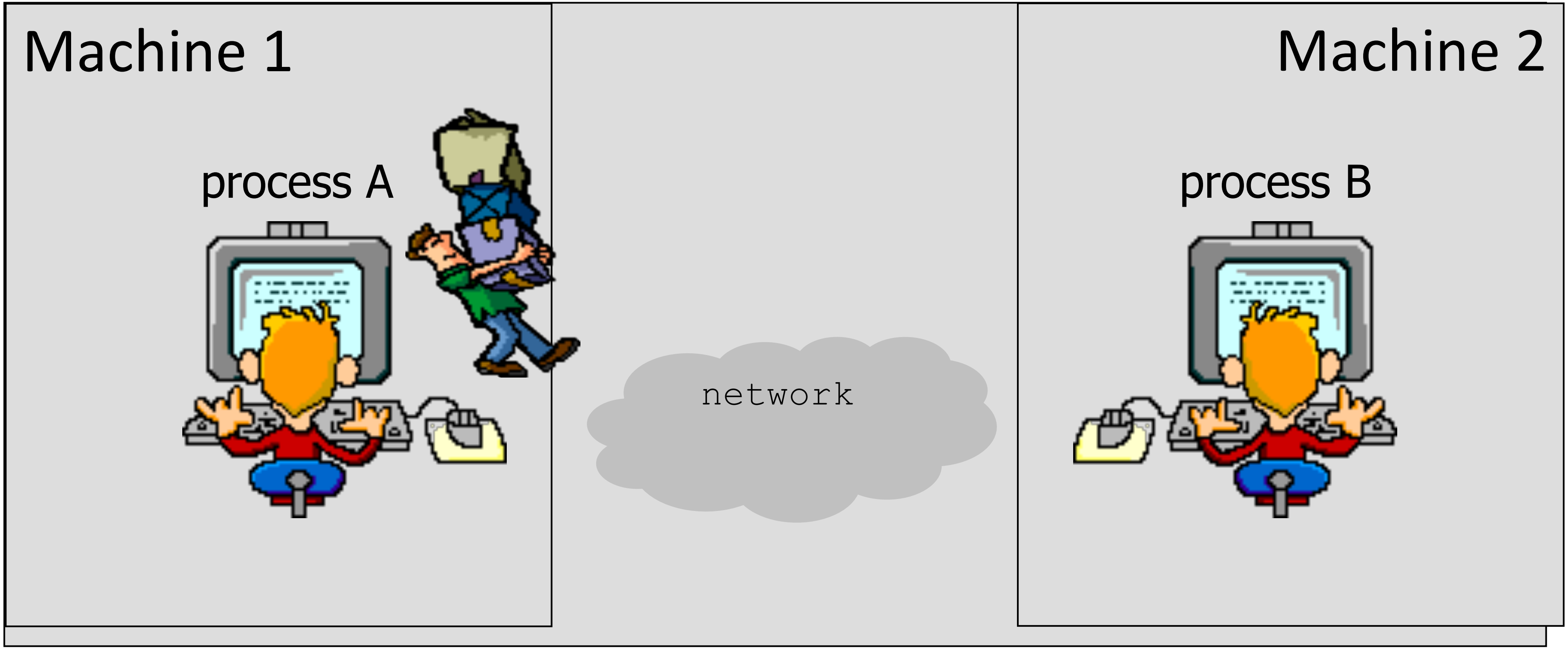
Port usage

- Popular applications have “well-known ports”
  - E.g., port 80 for Web and port 25 for e-mail
  - Well-known ports listed at http://www.iana.org
- Well-known vs. ephemeral ports
  - Server has a well-known port (e.g., port 80)
- By convention, between 0 and 1023; privileged
  - Client gets an unused “ephemeral” (i.e., temporary) port
  - By convention, between 1024 and 65535
- Flow identification
  - The two IP addresses plus the two port numbers
    - Sometimes called the “four-tuple”
  - Underlying transport protocol (e.g., TCP or UDP)
  - The “five-tuple”

Ports (summary)

- Not related to the physical architecture of the computer
- Just a number maintained by the operating systems to identify the end point of a connection

Big picture



Machine 1

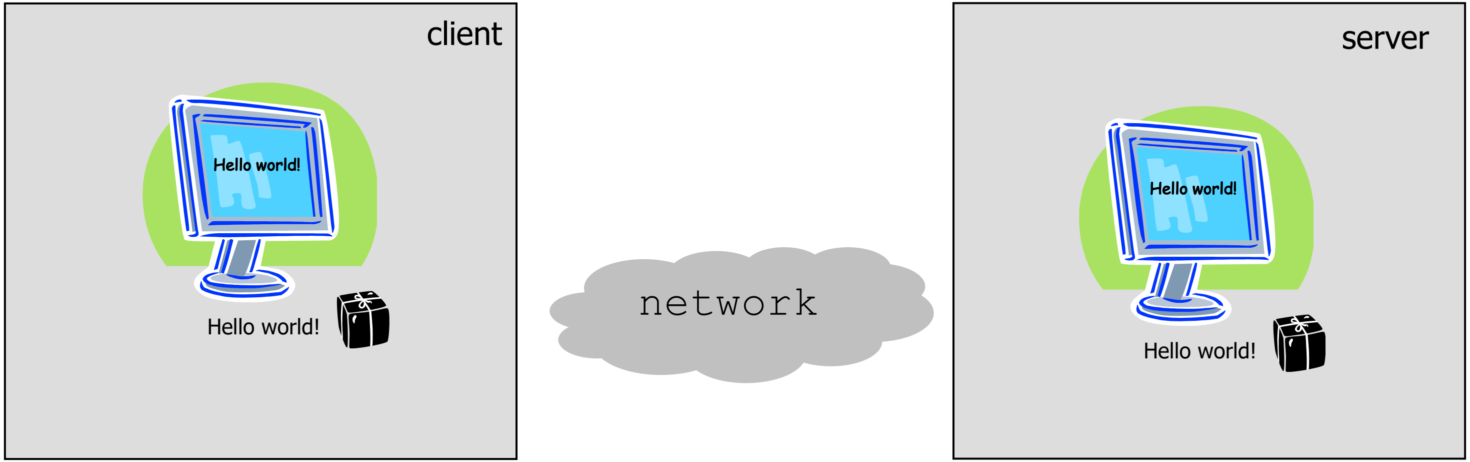
process A

network

Machine 2

process B

What we want?


client
Hello world!
Hello world!
network
server
Hello world!
Hello world!

What we want

Client

```
int main()
{
  char buf[13];

  /* Send data */
  write(sd, “Hello world!”, 12);

  /* Read data from the socket */
  read(sd, buf, 12);

  /* Add a string termination sign,
     and write to the screen. */
  buf[12] = ‘\0’;
  printf(“%s\n”, buf);


}
```

Server

```
int main()
{
  char buf[13];


  /* read data from the sd and
     write it to the screen */
  read(sd, buf, 12);
  buf[12] = ‘\0’;
  printf(“%s\n”, buf );

  /* send data back over the connection */
  write(sd, buf, 12);


}
```

Read & Write

- Same functions used for files etc.

- The call **read**(sd, buffer, n);
  - Reads *up to* n characters
  - From socket sd
  - Stores them in the character array buffer

- The call **write**(sd, buffer, n);
  - Writes *up to* n characters
  - From character array buffer
  - To the socket sd

Alternatives to Read & Write

- The call **recv**(sd, buffer, n, flags);
  - Reads *up to* n characters
  - From socket sd
  - Stores them in the character array buffer
  - Flags, normally just 0, but e.g., MSG_DONTWAIT, MSG_MORE,…
- The call **send**(sd, buffer, n, flags);
  - Writes *up to* n characters
  - From character array buffer
  - To the socket sd
  - Flags
- Several similar functions like …to/from, …msg

Creation of a connection

- One side must be the active one
  - take the initiative in creating the connection
  - this side is called the ***client***

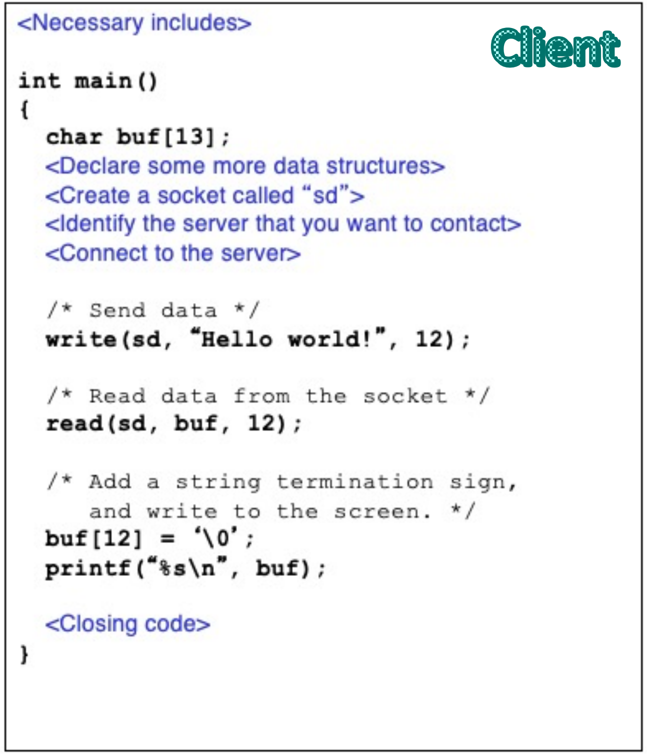
- The other side must be passive
  - it is prepared for accepting connections
  - waits for someone else to take initiative for creating a connection
  - this side is called the ***server***

- From now: server is a **process,** not a machine

Special for the server side

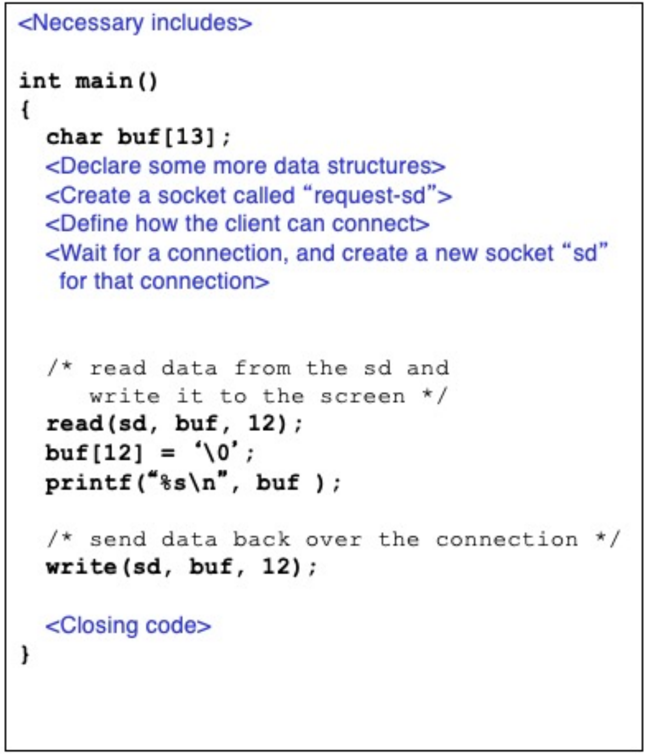
- In case of **TCP**
  - one socket on the server side is dedicated to waiting for a connection
  - for each client that takes the initiative, a separate socket on the server side is created
  - this is useful for all servers that must be able to serve several clients concurrently (web servers, mail servers, …)

To do – slightly more details



Client

```
<Necessary includes>

int main()
{
  char buf[13];
  <Declare some more data structures>
  <Create a socket called “sd”>
  <Identify the server that you want to contact>
  <Connect to the server>

  /* Send data */
  write(sd, “Hello world!”, 12);

  /* Read data from the socket */
  read(sd, buf, 12);

  /* Add a string termination sign,
     and write to the screen. */
  buf[12] = ‘\0’;
  printf(“%s\n”, buf);

  <Closing code>
}
```



Server

```
<Necessary includes>

int main()
{
  char buf[13];
  <Declare some more data structures>
  <Create a socket called “request-sd”>
  <Define how the client can connect>
  <Wait for a connection, and create a new socket “sd”
   for that connection>


  /* read data from the sd and
     write it to the screen */
  read(sd, buf, 12);
  buf[12] = ‘\0’;
  printf(“%s\n”, buf );

  /* send data back over the connection */
  write(sd, buf, 12);

  <Closing code>
}
```

<Necessary includes>

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
```

- These five files are needed by both client and server
- They include definitions and declarations as described on the following sides
- man-pages will have the info you need

<Necessary includes>

#include <netinet/in.h>	• Prototype & defines (htons, etc.) • sockaddr in
#include <sys/socket.h>	• Prototypes(send, connect, etc.)
#include <netdb.h>	• Prototypes (gethostbyname, etc.)
#include <stdio.h>	• Prototypes (printf, etc.)
#include <string.h>	• Prototypes(memset, etc.)

Create a socket

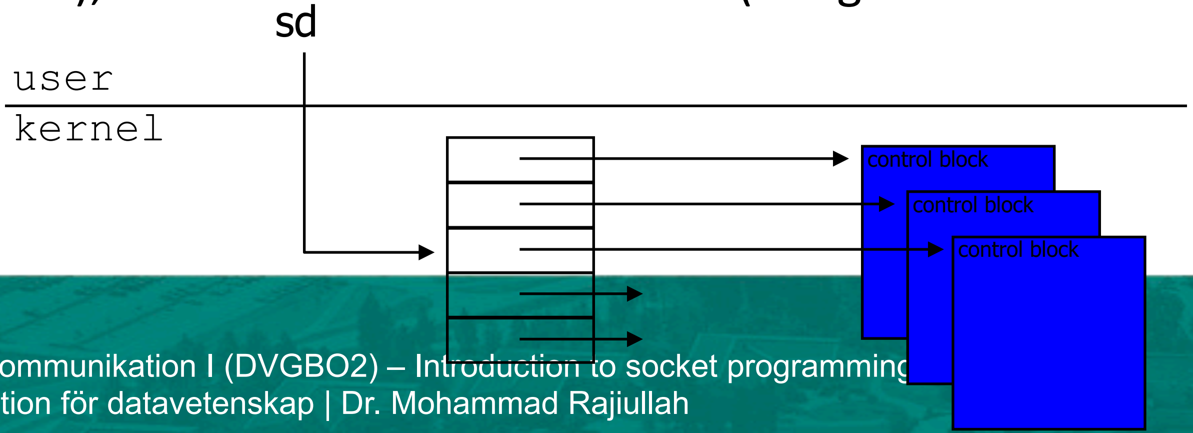
Client

```
/* declarations */
int sd;

/* creation of the socket */
sd = socket(PF_INET,
               SOCK_STREAM,
               IPPROTO_TCP);
```

Server

```
/* declarations */
int request_sd;

/* creation of the socket */
request_sd = socket(PF_INET,
                      SOCK_STREAM,
                      IPPROTO_TCP);
```

- Call to the function **socket()** creates a transport control block (hidden in kernel), and returns a reference to it (integer used as index)



More about the socket call

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

- PF_INET, SOCK_STREAM and IPPROTO_TCP are constants that are defined in the included files

- The use of the constants that we used on the previous slides (and above) creates a *TCP socket*

- Many other possibilities exist
  - Domain: PF_UNIX, PF_INET, PF_INET6, …
  - Type: SOCK_STREAM, SOCK_DGRAM, …
  - Protocol: IPPROTO_TCP, IPPROTO_UDP, …
- protocol can be NULL, OS choses apropriate proocol (use with care!)

How to identify clients to accept, and servers to contact?

- Machine??
  - by its IP address (e.g., 129.240.65.59)

- Application/service/program??
  - by (IP address and) port number
  - standard applications have own, “well-known” port numbers
    - SSH: 22
    - Mail: 25
    - Web: 80
    - Look in /etc/services for more

Address structure

- struct **sockaddr_in** :
  - sin_family address family used (defined through a macro)
  - sin_port 16-bit transport protocol port number
  - sin_addr 32-bit IP address defined as a new structure in_addr having one s_addr element only
  - sin_zero padding (to have an equal size as sockaddr)
  - declared in <netinet/in.h>
- Defines IP address and port number in a way the Berkeley socket API needs it
- man 7 ip

Address structure

Client

```
/* declaration */
struct sockaddr_in serveraddr;

/* clear the structure */
memset(&serveraddr, 0,
      sizeof(struct sockaddr_in));

/* This will be an address of the
 * Internet family */
serveraddr.sin_family = AF_INET;

/* Add the server address –*/
inet_pton(AF_INET,
          “xxx.xxx.xxx.xxx”,
          &serveraddr.sin_addr);

/* Add the port number */
serveraddr.sin_port = htons(2009);
```

Server

```
/* declaration */
struct sockaddr_in serveraddr;

/* clear the structure */
memset(&serveraddr, 0,
      sizeof(struct sockaddr_in));

/* This will be an address of the
 * Internet family */
serveraddr.sin_family = AF_INET;

/* Allow all own addresses to receive */
serveraddr.sin_addr.s_addr = INADDR_ANY;

/* Add the port number */
serveraddr.sin_port = htons(2009);
```

Address structure

- Fill address type (“family”), address and p
  - serveraddr.sin_family = AF
  - server

Why not only:

- serveraddr.sin_addr.s_addr = 129.240.65.59 ?
- serveraddr.sin_port = 2009 ?

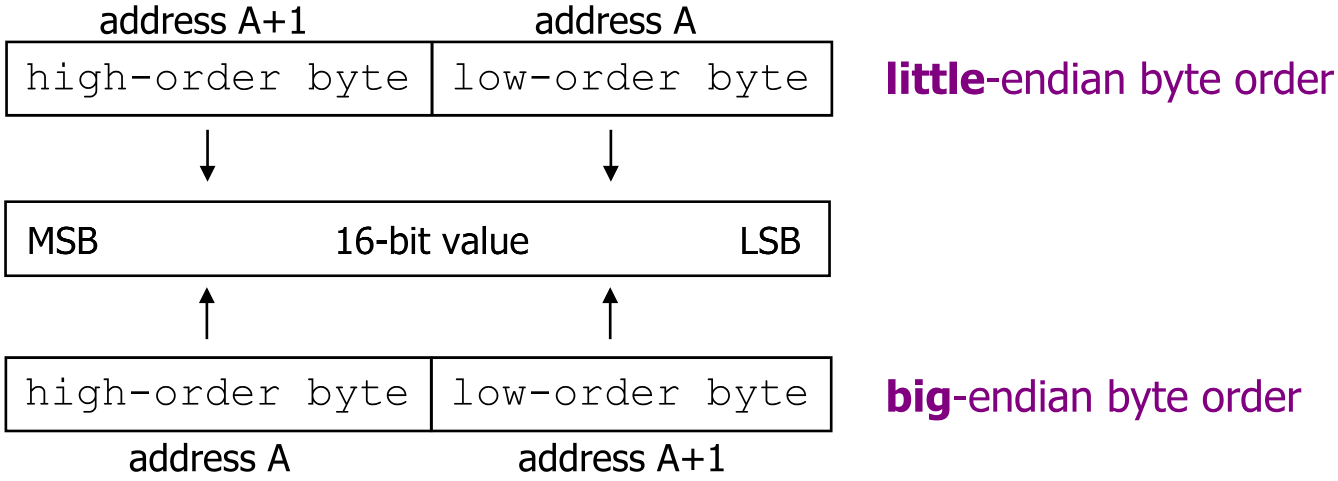
r); (@ client)

ddr.sin_port = htons(2009);

  - AF_INET
    - a constant indicating that Internet protocols will be used
  - INADDR_ANY
    - a constant meaning any (Internet) address
    - in this context: any own Internet address

Byte order

- Different machines may have different representation of multi-byte values
- Consider a 16-bit integer: made up of 2 bytes

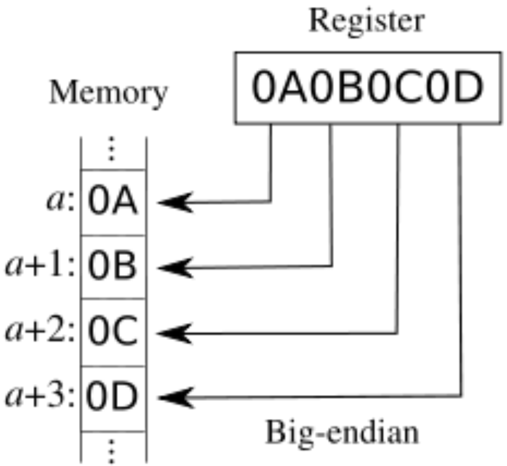


Byte Order: Storing 32-bit 0x0A0B0C0D

- Assuming 8-bit (one byte) atomic elements…
- **…big endian:**
  - the most significant byte (MSB), 0x0A, is stored on the *lowest* memory address
  - the least significant byte (LSB), 0x0D, is stored on the **highest** memory address

increasing memory addresses →

…	0x0A	0x0B	0x0C	0x0D	…

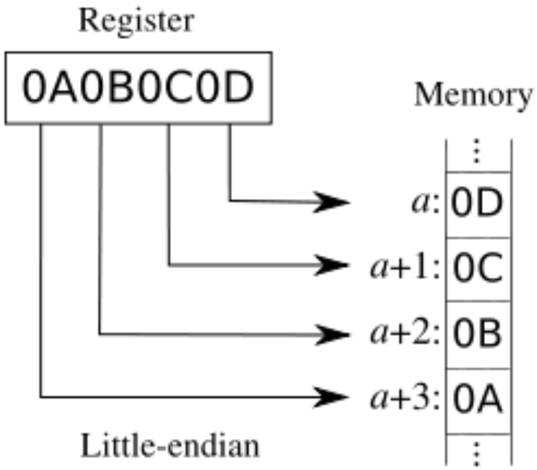


Byte Order: Storing 32-bit 0x0A0B0C0D

- **… little endian:**
  - 0x0A is stored on the **highest** memory address
  - 0x0D is stored on the *lowest* memory address

increasing memory addresses →

…	0x0D	0x0C	0x0B	0x0A	…



Byte Order: IP address example

- IPv4 host address: represents a 32-bit address
  - written on paper (”dotted decimal notation”): **129.240.71.213**
  - binary in bits: **10000001 11110000 01000111 10001011**
  - hexadecimal in bytes: **0x81 0xf0 0x47 0x8b**

Problem!

- Big-endian (”normal” left to right):
  - one 4 byte int on PowerPC, POWER, Sparc, …: 0x**81f0478b**
- Little-endian:
  - one 4 byte int on x86, StrongARM, XScale, …: 0x**8b47f081**
- **Network byte order**: 0x**81f0478b**

Byte Order: Translation

- Byte order translation makes communication over several platforms possible
- htons() / htonl()
  - host-to-network short / long
  - translate a 16 / 32-bit integer value to network format
- ntohs() / ntohl()
  - network-to-host short/long
  - translate a 16 / 32-bit integer value to host format

- Little-endian (x86 etc.): ntohl(**0x81f0478b**) == **0x8b47f081**
- Big-endian (PowerPC etc.): ntohl(**0x81f0478b**) == **0x81f0478b**

Presentation and Numeric Address Formats

- The network…
  - …does not interpret the “dotted decimal notation” *presentation* format
  - …needs a *numeric* binary format in network byte order
- inet_pton()
  - translate the text string to a numeric binary format needed by the address structure
- inet_ntop()
  - translate the e to a text string

`inet_pton()` is new for IPv6.

Oldest:

```
serveraddr.sin_addr.s_addr =
     inet_addr(“129.240.65.59”);
```

Newer:

```
inet_aton(“129.240.65.59”,
          &serveraddr.sin_addr);
```

Client

```
✓<Necessary includes>

int main()
{
   char buf[13];
✓  <Declare some more data structures>
✓  <Create a socket called “sd”>
✓  <Identify the server that you want to contact>
   <Connect to the server>

   /* Send data */
   write(sd, “Hello world!”, 12);

   /* Read data from the socket */
   read(sd, buf, 12);

   /* Add a string termination sign,
      and write to the screen. */
   buf[12] = ‘\0’;
   printf(“%s\n”, buf);

   <Closing code>
}
```

Server

```
✓<Necessary includes>

int main()
{
   char buf[13];
✓  <Declare some more data structures>
✓  <Create a socket called “request-sd”>
✓  <Define how the client can connect>
   <Wait for a connection, and create a new socket “sd”
    for that connection>


   /* read data from the sd and
      write it to the screen */
   read(sd, buf, 12);
   buf[12] = ‘\0’;
   printf(“%s\n”, buf );

   /* send data back over the connection */
   write(sd, buf, 12);

   <Closing code>
}
```

Client

```
/* Connect */
connect(    sd,
            (struct sockaddr*)&serveraddr,
            sizeof(struct sockaddr_in));
```

Server

```
/* Bind the address to the socket */
bind(request_sd,
     (struct sockaddr*)&serveraddr,
     sizeof(struct sockaddr_in);


/* Activate listening on the socket */
listen(request_sd, SOMAXCONN);


/* Wait for connection */
clientaddrlen =
             sizeof(struct sockaddr_in);

sd = accept(request_sd,
          (struct sockaddr*)&clientaddr,
          &clientaddrlen);
```

Some details about the previous slides

- **bind(** int sfd, struct sockaddr *a, socklen_t al **)**
  - a machine can have several addresses (several network cards, loopback, …) – "assign a name"
  - tells the socket on the server side which local protocol (i.e., *IP address* and *port number)* to listen to

- **listen(** int sfd, int backlog **)**
  - prepares the server for listening to connect requests, and initializes a queue for connect requests (→ passive)
  - the second parameter (SOMAXCONN) defines how long the queue(s) should be

More details

- sd = **accept(** int sfd, struct sockaddr *a, socklen_t *al **)**
  - take the first connect request from the connect request queue
  - wait for the connect request to arrive if the queue is empty
  - returns a ***new socket*** that the server can use to communicate with the client
  - a (clientaddr) contains information about the client
  - al must be initialized, so accept knows size of a

- **connect(** int sfd, struct sockaddr *serv_a, socklen_t al **)**
  - connects client socket to a server that is specified in the address structure
  - a three-way handshake is initiated for TCP
  - possible errors
    - ETIMEDOUT – no response (after several tries) and timer expired
    - ECONNREFUSED – server not running or not allowed to connect
    - EHOSTUNREACH – HOST not reachable
    - ENETUNREACH – NET not reachable

Closing of sockets

Client

```
/* Close the socket */
close(sd);
```

Server

```
/* Close both sockets */
close(sd);
close(request_sd);
```

- Note that the semantics of close depends
  - On the kind of protocol
  - Some possible extra settings
  - (similar for file descriptors used to operate on disk…
- All data that has not been read yet may be thrown away

Client

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>

int main()
{
  /* Declarations */
  struct sockaddr_in serveraddr;
  int sd;
  char buf[13];

  /* Create socket */
  sd = socket(PF_INET,
              SOCK_STREAM,
              IPPROTO_TCP);

  /* Clear address structure */
  memset(&serveraddr, 0,
        sizeof(struct sockaddr_in));

  /* Add address family */
  serveraddr.sin_family = AF_INET;
```

Client ctd.

```
  /* Add IP address of server*/
  inet_pton(AF_INET, “xx.xx.xx.xx”,
            &serveraddr.sin_addr);
  /* Add the port number */
  serveraddr.sin_port = htons(2009);

  /* Connect */
  connect(sd,
          (struct sockaddr*)&serveraddr,
          sizeof(struct sockaddr_in));

  /* Send data */
  write(sd, “Hello world!”, 12 );

  /* Read data */
  read(sd, buf, 12 );

  /* add string end sign, write to screen*/
  buf[12] = ‘\0’;
  printf(“%s\n”, buf);

  /* Close socket */
  close(sd);
}
```

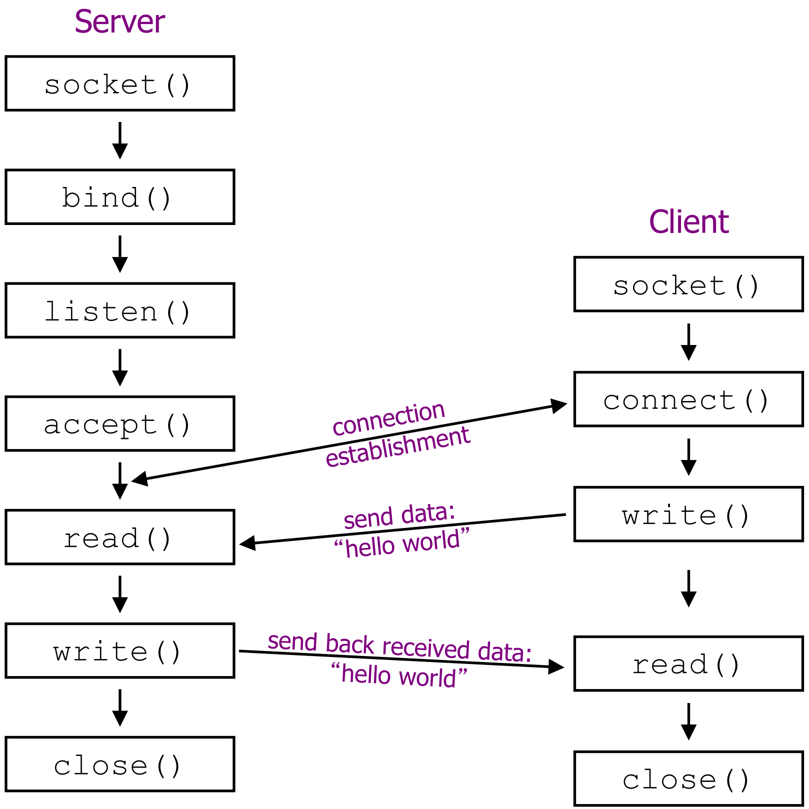
Server

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>

int main()
{
  /* Declarations */
  struct sockaddr_in serveraddr;
  struct sockaddr_in clientaddr;
  int clientaddrlen;
  int request_sd, sd;
  char buf[13];

  /* Create socket */
  request_sd = socket(PF_INET,
                      SOCK_STREAM,
                      IPPROTO_TCP);

  /* Fill in the address structure */
  memset(&serveraddr, 0,
        sizeof(struct sockaddr_in));
  serveraddr.sin_family = AF_INET;
  serveraddr.sin_addr.s_addr = INADDR_ANY;
  serveraddr.sin_port = htons(2009);
```

Server ctd.

```
  /* Bind address to socket */
  bind(request_sd,
       (struct sockaddr*)&serveraddr,
       sizeof(struct sockaddr_in));

  /* Activate connect request queue */
  listen(request_sd, SOMAXCONN);

  /* Receive connection */
  clientaddrlen =
      sizeof(struct sockaddr_in);
  sd = accept(request_sd,
            (struct sockaddr*)&clientaddr,
            &clientaddrlen);

  /* Read data from socket and write it */
  read(sd, buf, 12);
  buf[12] = ‘\0’;
  printf(“%s\n”, buf);

  /* Send data back over connection */
  write(sd, buf, 12);

  /*Close sockets */
  close(sd); close(request_sd);
}
```

Summary of
Socket Functions for our Elementary TCP Client-Server



Compilation of these socket programs

Server

```
...

int main()
{
  /* Declarations */
  ...


  /* Create socket */
  request_sd = socket(...);

  /* Fill in the address structure */
  ...

  /* Bind address to socket */
  bind(...);

  /* Activate connect request queue */
  listen(...);
```

Server ctd.

```
  /* Receive connection */
  sd = accept(...);

  /* Process  the request*/
  ...

  /*Close sockets */
  close(sd);









  close(request_sd);
}
```

Iterative servers?

Server

```
...

int main()
{
  /* Declarations */
  ...


  /* Create socket */
  request_sd = socket(...);

  /* Fill in the address structure */
  ...

  /* Bind address to socket */
  bind(...);

  /* Activate connect request queue */
  listen(...);
```

Server ctd.

```
  for (;;) {
    /* Receive connection */
    sd = accept(...);

    /* Process  the request*/
    ...









    /*Close sockets */
    close(sd);
  }

  close(request_sd);
}
```

Concurrent servers?

Server

```
...

int main()
{
  /* Declarations */
  ...
  pid_t pid;

  /* Create socket */
  request_sd = socket(...);

  /* Fill in the address structure */
  ...

  /* Bind address to socket */
  bind(...);

  /* Activate connect request queue */
  listen(...);
```

Server ctd.

```
  for (;;) {
    /* Receive connection */
    sd = accept(...);

    if ((pid = fork()) == 0) {
       close(request_sd);
      /* Process  the request*/
      ...

      /*Close sockets */
      close(sd);
      exit(0)
    }

    /*Close sockets */
    close(sd);
  }

  close(request_sd);
}
```

Summary

- We have implemented a short program where two processes communicate over a network