INF1060:
Introduction to Operating Systems and Data Communication

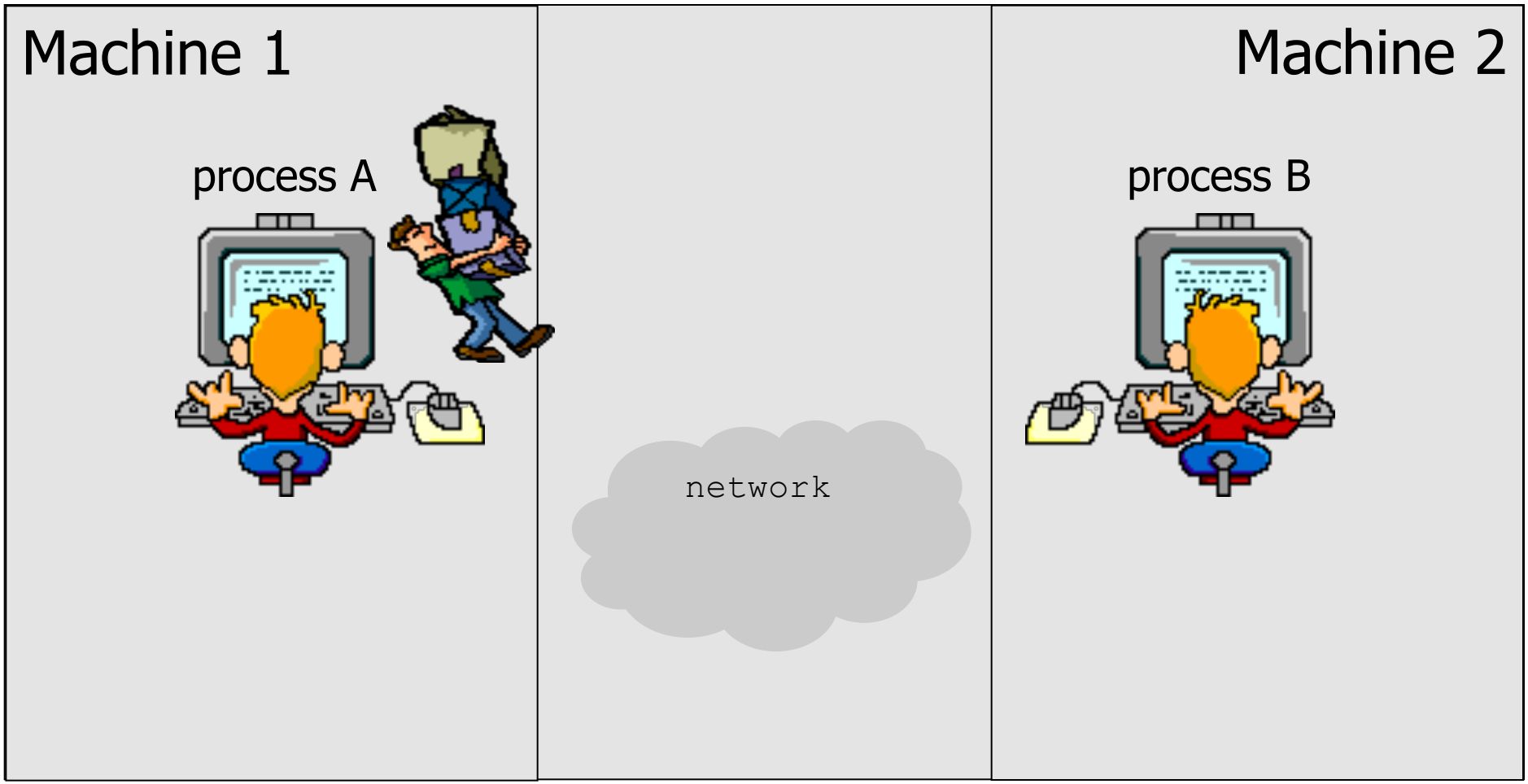
Data Communication:

Introduction to Berkeley Sockets

Michael Welzl

(adapted from lectures by Pål Halvorsen, Carsten Griwodz & Olav Lysne)

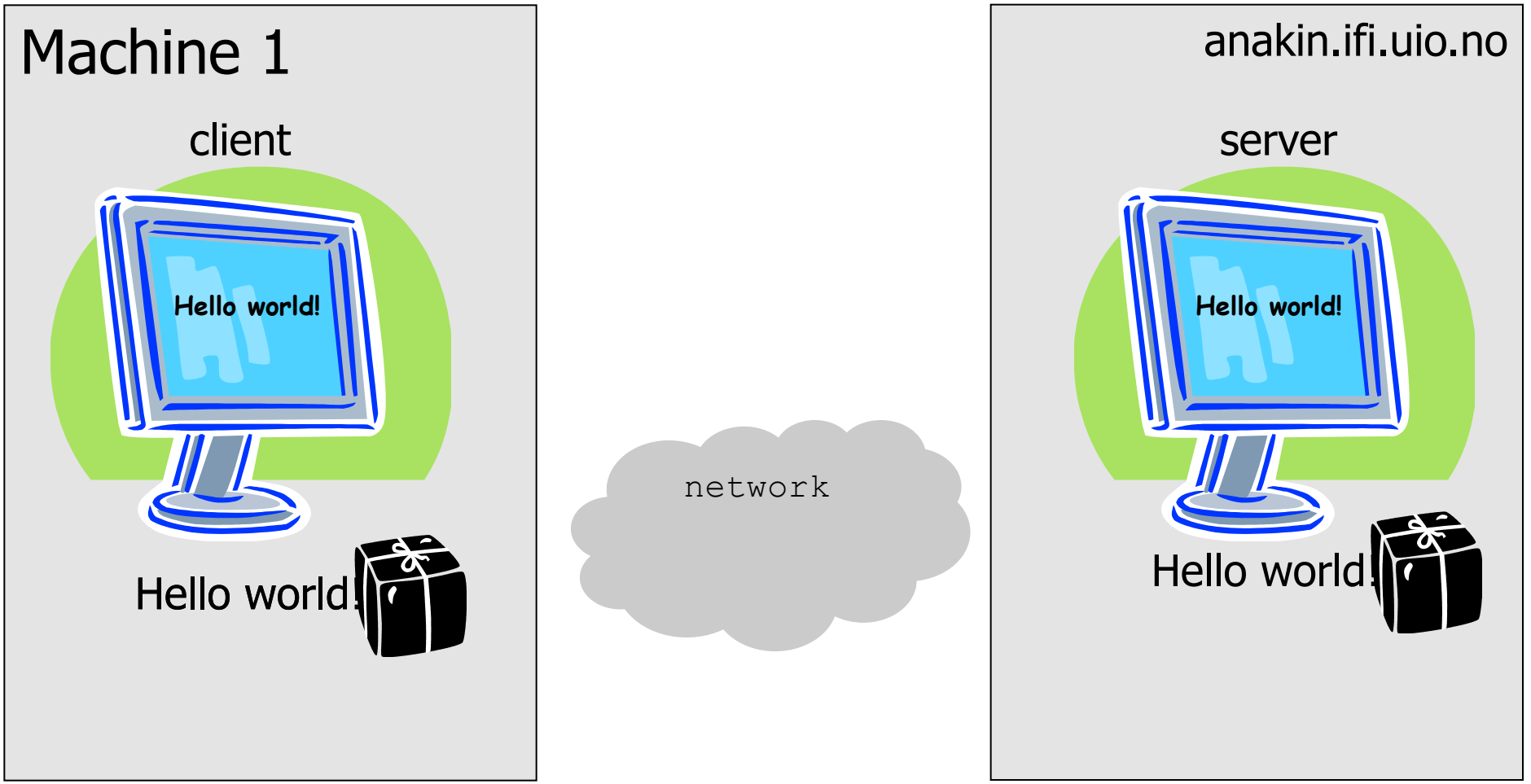
Big Picture



Goal

- Introduce socket API
- We will write two programs
  - A “client” and a “server”
- Each will run on one machine
  - the server will run on “anakin.ifi.uio.no” (129.240.64.199)
- 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 we want


Machine 1
client
Hello world!
Hello world!
network
anakin.ifi.uio.no
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 `n` characters
  - From socket `sd`
  - Stores them in the character array `buffer`
- The call `write(sd, buffer, n);`
  - Writes `n` characters
  - From character array `buffer`
  - To the socket `sd`

Alternatives to Read & Write

- The call `recv(sd, buffer, n, flags);`
  - Reads `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 `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***
- This use of the words client and server is not entirely consistent with everyday use, but for programming this is conventional

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>
```

- prototypes & defines (`htons`, etc.)
- `sockaddr in`
- prototypes (`send`, `connect`, etc.)
- prototypes (`gethostbyame`, etc.)
- prototypes (`printf`, etc.)
- prototypes (`memset`, etc.)

- These five files are needed by both client and server
- They include definitions and declarations as described on the following sides
- Some systems will have the same declarations in different files – the above examples should work at IFI (see `/usr/include` on Linux & Solaris)

<Create a socket>

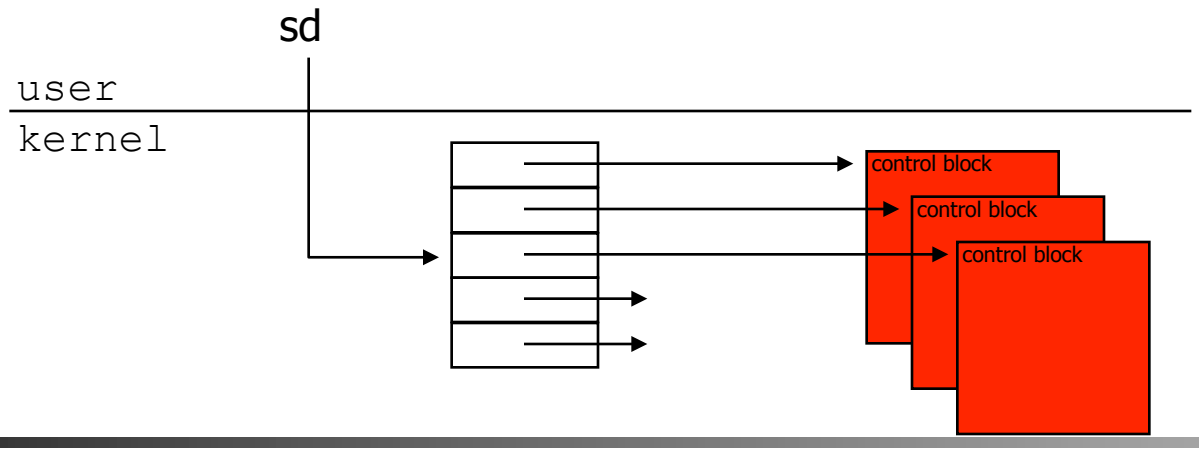
Client

```
/* 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
  - `<bits/socket.h>` which is included by `<sys/socket.h>`
  - `<netinet/in.h>`
- 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`, …

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

- Machine??
  - by its IP address (e.g., 129.240.64.199)
- 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

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 - anakin */
inet_pton(AF_INET,
          "129.240.64.199",
          &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);
```

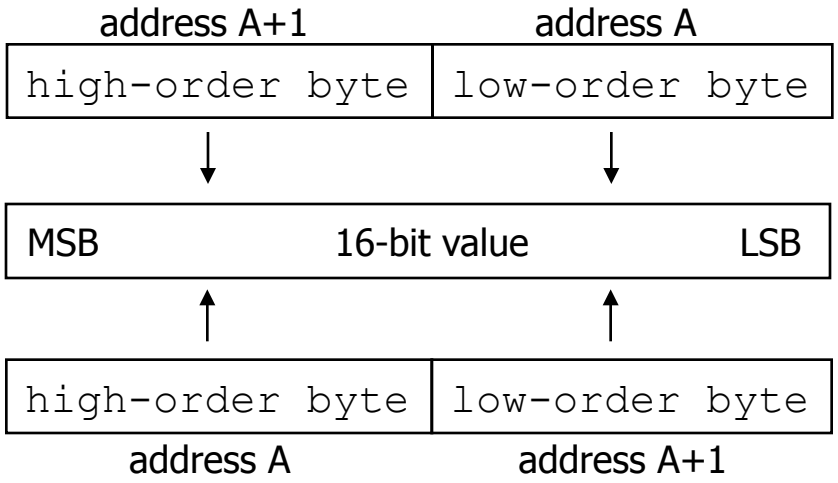
Address structure

- Fill address type ("family"), address and port number into the structure
  - `serveraddr.sin_family = AF_INET;`
  - `serveraddr.sin_addr.s_addr = INADDR_ANY;` (@ server)
  - `inet_pton( AF_INET, "129.240.64.199", &serveraddr.sin_addr );` (@ client)
  - `serveraddr.sin_port = htons( 2009 );`
  - AF_INET
    - a co
    - ...ning any (Internet) address
    - in this context: any own Internet address

Why not only:
- `serveraddr.sin_addr.s_addr = 129.240.64.199` ?
- `serveraddr.sin_port = 2009` ?

Byte Order

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

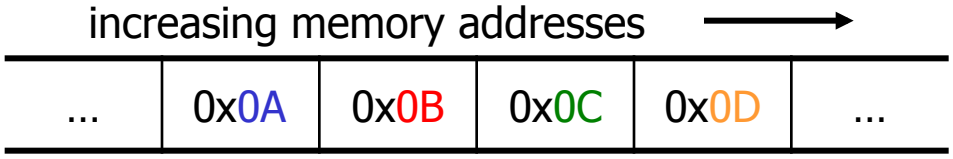


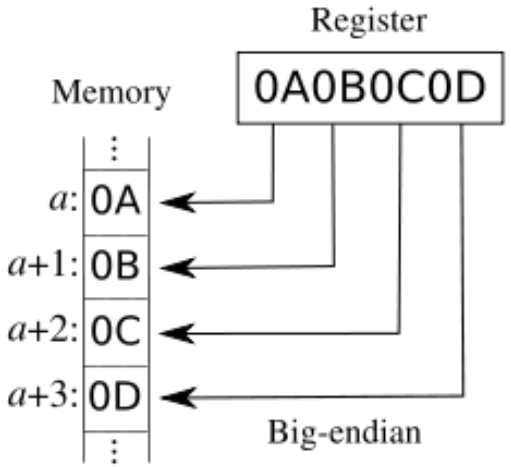
little-endian byte order

big-endian byte order

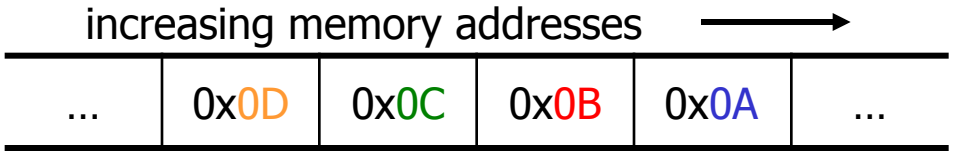
Byte Order: Storing 32-bit 0x0A0B0C0D

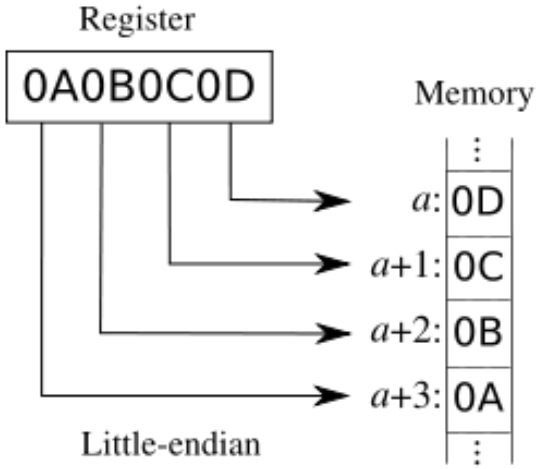
- Assuming 8-bit (one byte) atomic elements…
- **…big endian:**
  - the most significant byte (MSB), 0x0A, is stored on the *lowest* memory address
  - the least significant byte (LSB), 0x0D, is stored on the **highest** memory address





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





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`

- Big-endian ("normal" left to right):
  - one 4 byte `int` on PowerPC, POWER, Sparc, ...: 0x81f0478b
- Little-endian:
  - one 4 byte `int` on x86, StrongARM, XScale, ...: 0x8b47f081

Problem!

- Middle/mixed/PDP endian:
  - one 4 byte int on PDP-11: 0xf0818b47
- **Network byte order**: 0x81f0478b

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 (numeri… …a text string

`inet_pton()` is new for IPv6.

Oldest:

```
serveraddr.sin_addr.s_addr =
          inet_addr("129.240.64.199");
```

Newer:

```
inet_aton("129.240.64.199",
          &serveraddr.sin_addr);
```

How far have we gotten now?

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>

```
}
```

Binding, Listening, Accepting and Connecting

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

Complete Client

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 anakin.ifi.uio.no */
  inet_pton(AF_INET, "129.240.64.199",
            &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);
}
```

Complete Server

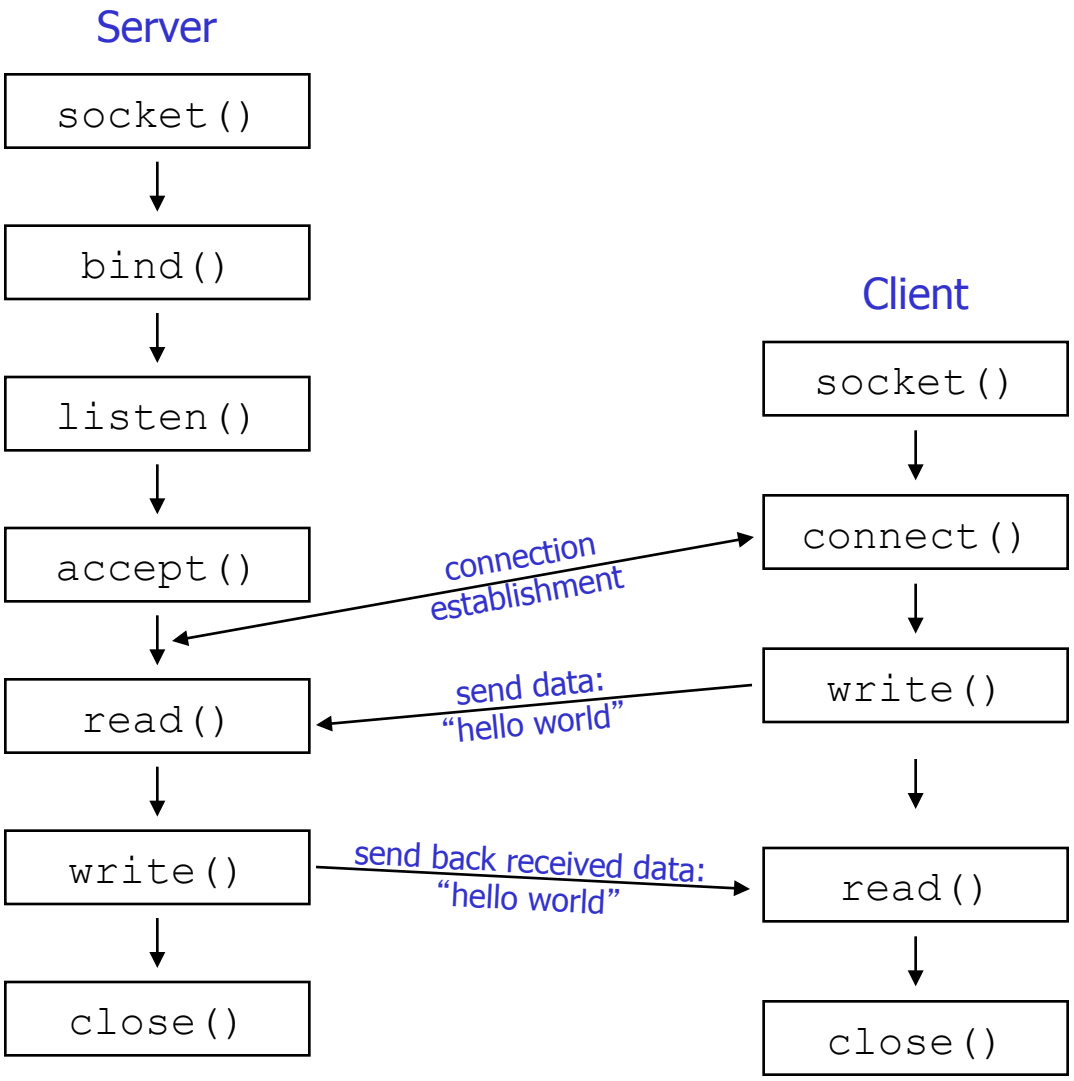
Server

```
#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

- The example can be downloaded from the web pages (http://www.ifi.uio.no/~inf1060/programs/client-server-example)
- IFI's Linux machines
  - `gcc client1.c -o client`
- IFI's Solaris machines
  - `gcc client1.c -o client -lsocket -lnsl`
- Cygwin on Windows
  - `gcc client1.c -o client`
- Similar for `server1.c`
- For testing, run server on anakin (or change the address in the client) and start client on another machine
  - Testing on one host: use 127.0.0.1
- Note for BSD / Mac systems: `#include <sys/types.h>`

Complete Server

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?

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?

Concurrent Iterative 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);
}
```

Select

- Problems with these examples:
  - iterative: cannot serve more than one socket at once
  - concurrent: overhead (a process per socket)

- Solution: functions that tell you when a socket becomes available (`select, poll`)

- `int select(int nfds, fd_set *restrict readfds, fd_set *restrict writefds,fd_set *restrict errorfds, struct timeval *restrict timeout)`
  - check whether fd's (sockets) from the `nfds` set are available for reading (`readfds`), writing (`writefds`), or have exceptional conditions pending (`errorfds`)
  - Null argument: don't check. Timeout = time limit for check (`Null` = block).
  - result is given by changing `readfds` / `writefds` / `errorfds`

Select usage and macros

- Select usage
  - Declare and initialize fd_set; add relevant sockets to fd_set; give select a copy of fd_set for every operation of interest (read/write/exceptional); loop through copies to take action

- Preparing fd_set is done with some macros
  - **`FD_CLR(fd, &fdset)`**
    - removes the socket descriptor `fd` from the socket descriptor set `fdset`
  - **`FD_ISSET(fd, &fdset)`**
    - returns nonzero if socket descriptor `fd` is a member of `fdset`; else 0
  - **`FD_SET(fd, &fdset)`**
    - adds socket descriptor `fd` to `fdset`
  - **`FD_ZERO(&fdset)`**
    - initializes `fdset` to 0, representing the empty set
  - **`FD_SETSIZE`** - max. number of FDs; use this as the first parameter for select

Complete Select-based Server

Test with e.g. two clients!

Server

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

int main()
{
  /* Declarations */
  struct sockaddr_in serveraddr;
  struct sockaddr_in clientaddr;
  int clientaddrlen, i, rc;
  int request_sd, sd[2], numsocks, maxsocks;
  char buf[13];
  fd_set fds, readfds;
  struct timeval timeout;

  numsocks = 0; maxsocks = 2;
  timeout.tv_sec = 20;
  timeout.tv_usec = 0;

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

Server ctd.

```
  /* 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);

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

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

  /* Initialize fd set */
  FD_ZERO(&fds);
  FD_SET(request_sd, &fds);
```

Complete Select-based Server ctd.

Server ctd.

```
for (;;) {

  readfds=fds;
  rc=select(FD_SETSIZE, &readfds, NULL,
                  NULL, &timeout);

  /* Something went wrong */
  if (rc<0)
    return -1;

  /* Nothing happened,select continued */
  if (rc==0) {
    printf("Timeout!\n");

    for(i=0; i<numsocks; i++) {
      /* Send a response */
      write(sd[i], "Server ACK!",11);
      /* Close sockets */
      close(sd[i]);
      FD_CLR(sd[i], &fds);

    }
    return 0;

  }
```

Server ctd.

```
    for (i = 0; i < FD_SETSIZE; i++)
      if(FD_ISSET (i, &readfds)) {

        if(i == request_sd) {
          /* new connection request */
          if(numsocks < maxsocks) {
            sd[numsocks] = accept(request_sd,
              (struct sockaddr *)&clientaddr,
              (socklen_t *)&clientaddrlen);
            FD_SET(sd[numsocks], &fds);
            numsocks++;
          } else {
            printf("Ran out of socket space.\n");
            return -1;
          }
        } else {
          /* data arrived on an existing socket */
          read(i, buf,12);
          buf[12] = '\0';
          printf("From socket %d: %s\n",i,buf);
        }
      }
  }
  close(request_sd);
}
```

Summary

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

- Next: the magic of how data is sent…

Literature

- "Berkeley UNIX System Calls and Interprocess Communication", Lawrence Besaw, University of Wisconsin
  - is available through the course web pages

- Many books:
  - Kurose/Ross, "Computer Networking: A Top-Down Approach Featuring the Internet", 2nd ed., Addison-Wesley
  - Andrew Tanenbaum, "Computer Networks", 4th ed., Prentice Hall
  - W. Richard Stevens, "Unix Network Programming – Networking APIs: Sockets and XTI", volume 1, 2nd ed., Prentice Hall