Tölvusamskipti / Computer Networks
T-409-TSAM
Háskólinn í Reykjavík

Hans P. Reiser

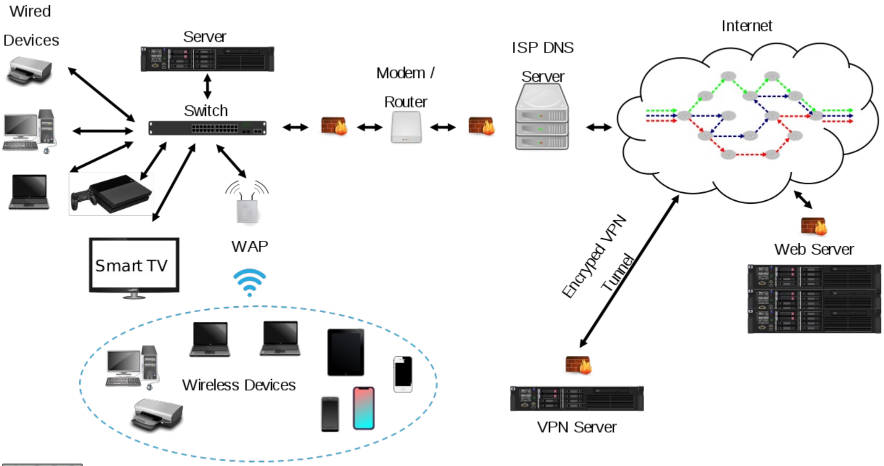
August 18th, 2022

* slides largely based on Stephan Schiffel´s TSAM slides

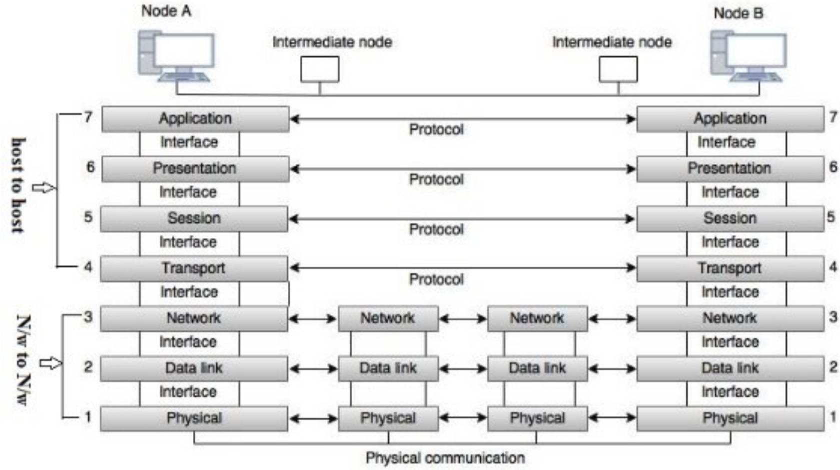
Outline

Introduction to layered network models

Computer Networks



OSI 7 Layer Model (1984): X.200



OSI X.200 specification of the model: https://www.itu.int/rec/T-REC-X.200-199407-I/en

OSI 7 Layer Model (1984): X.200

- Theoretical model, developed at Honeywell
- Separated out 7 layers for different aspects of communication
- Each layer performs clearly defined functions
- Minimise dependency (information flow) across the layers
- Each layer depends on the previous one
- In practice, only 4 layers were typically used (to large extent)
- ... giving rise to the simplified TCP/IP reference model

IT'S A

LAYER 8

PROBLEM

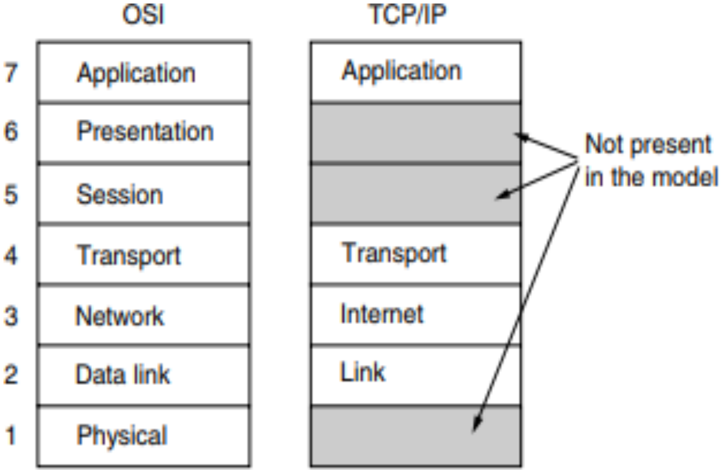


Figure 1-21. The TCP/IP reference model.

RFC 1122, Internet STD 3 (1989)	Cisco Academy[37]	Kurose,[38] Forouzan[39]	Comer,[40] Kozierok[41]	Stallings[42]	Tanenbaum[43]	Arpanet Reference Model (RFC 871)	OSI model
Four layers	*Four layers*	*Five layers*	*Four+one layers*	*Five layers*	*Five layers*	*Three layers*	*Seven layers*
"Internet model"	"Internet model"	"Five-layer Internet model" or "TCP/IP protocol suite"	"TCP/IP 5-layer reference model"	"TCP/IP model"	"TCP/IP 5-layer reference model"	"Arpanet reference model"	OSI model
Application	Application	Application	Application	Application	Application	Application/Process	Application
							Presentation
							Session
Transport	Transport	Transport	Transport	Host-to-host or transport	Transport	Host-to-host	Transport
Internet	Internetwork	Network	Internet	Internet	Internet		Network
Link	Network interface	Data link	Data link (Network interface)	Network access	Data link	Network interface	Data link
		Physical	(Hardware)	Physical	Physical		Physical

Terminology/Conventions

Computer Communication

- Requires a sender and a receiver
- Sender has to know who the receiver is
  - For Internet: Identification by IP address and port number
- Receiver has to be able to accept incoming connections
- Programming: both IP address and port number are wrapped up into "sockets"

(for now, we ignore special cases like multicast/broadcast, connectionless datagrams, etc.)

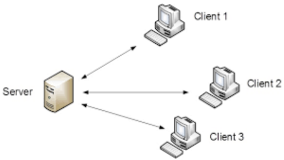
Client – Server Architecture

Client

- Initiates connection to the server
- Interfaces directly to the user
- Communicates with the server

Server

- Waits for connections from clients
- Provides services to the client
- Communications with the clients
- Handles many clients simultaneously

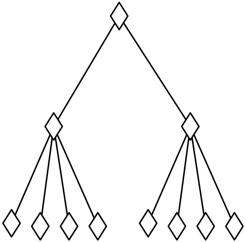


1:N (server:clients)

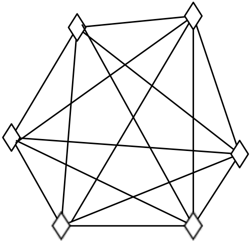
Peer-to-Peer:
Does both: M:N

- Some or all peers take on a server role as well
- Organized topologies tend to emerge at scale
- Software needs to be both a client and a server

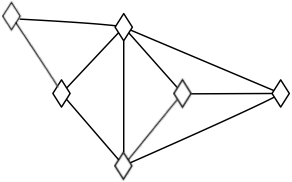
Topologies



Strictly Hierachical



Full Mesh



Partial Mesh

...

The `socket()` Interface

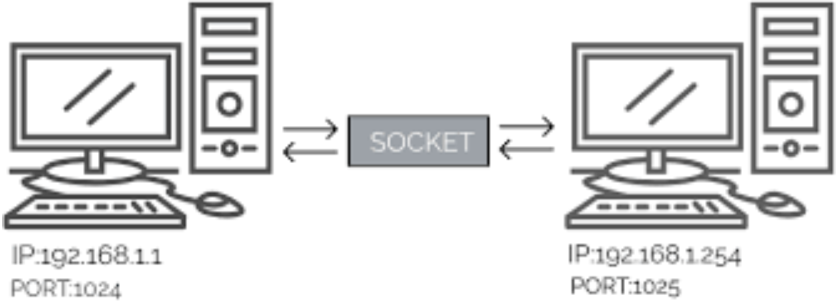
socket() Interface: History

- There are a lot of variants
- Berkley sockets 4.2BSD 1983
- Evolved into pretty much identical POSIX sockets
- Winsock (1992) based on Berkley sockets - diverged to handle Windows
- Windows Socket 2 architecture (2018)
- OSX sockets derive from Berkley 4.4

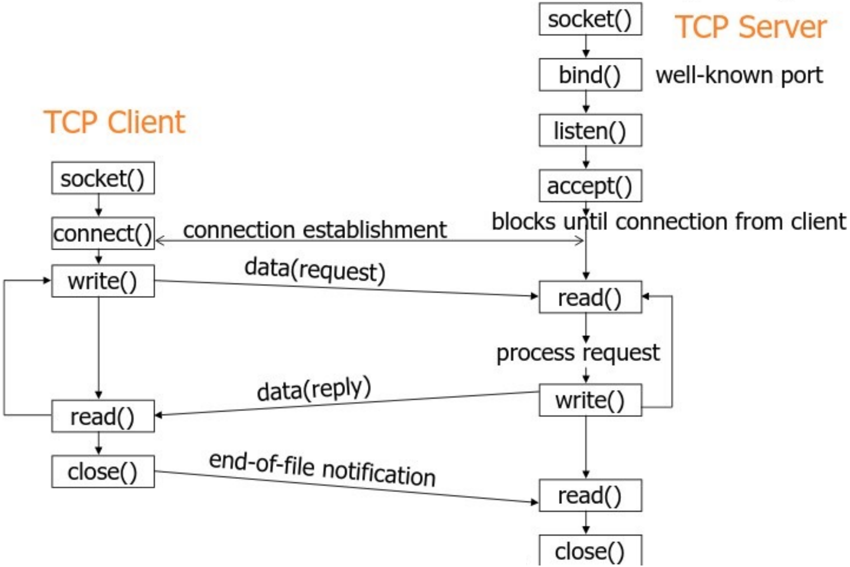
Berkeley Sockets API

Primitive	Meaning
Socket	**Create a new communication endpoint**
Bind	**Attach a local address to a socket**
Listen	**Announce willingness to accept connections**
Accept	**Block caller until a connection request arrives**
Connect	**Actively attempt to establish a connection**
Send	**Send some data over the connection**
Receive	**Receive some data over the connection**
Close	**Release the connection**

Socket Communication



Client-Server Communication (TCP)



Creating sockets

- First create a `socket` structure
- Then `bind()` the socket to a local address (IP, port)
  (usually not needed/wanted for the client side)
- Then use the socket to `connect()` to a remote machine
- or to `accept()` incoming connections

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

sock_fd = socket( domain , type , protocol )

bind( sock_fd , &sin , sizeof(sin))
```

socket(domain, type, protocol)

`int domain`	Protocol family: AF_INET [PF_INET is a synonym]
`int type`	Communication type. SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, etc.
`int protocol`	0 for default protocol
	see `/etc/protocols` for protocol numbers

- SOCK_STREAM — Sequenced, reliable 2-way stream (TCP)
- SOCK_DGRAM — Fixed max. length, unreliable message (UDP)
- SOCK_RAW — Raw network socket access

```
SOCKET(2)                 Linux Programmer's Manual                SOCKET(2)

NAME
       socket - create an endpoint for communication

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

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

DESCRIPTION
       socket()  creates  an  endpoint  for  communication  and returns a file
       descriptor that refers to that endpoint.  The file descriptor  returned
       by  a  successful  call will be the lowest-numbered file descriptor not
       currently open for the process.

       The domain argument specifies a communication domain; this selects  the
       protocol  family  which will be used for communication.  These families
       are  defined  in  <sys/socket.h>.   The  currently  understood  formats
       include:

       Name                Purpose                          Man page
       AF_UNIX, AF_LOCAL   Local communication              unix(7)
       AF_INET             IPv4 Internet protocols          ip(7)
       AF_INET6            IPv6 Internet protocols          ipv6(7)
       AF_IPX              IPX - Novell protocols
```

Relevant header files

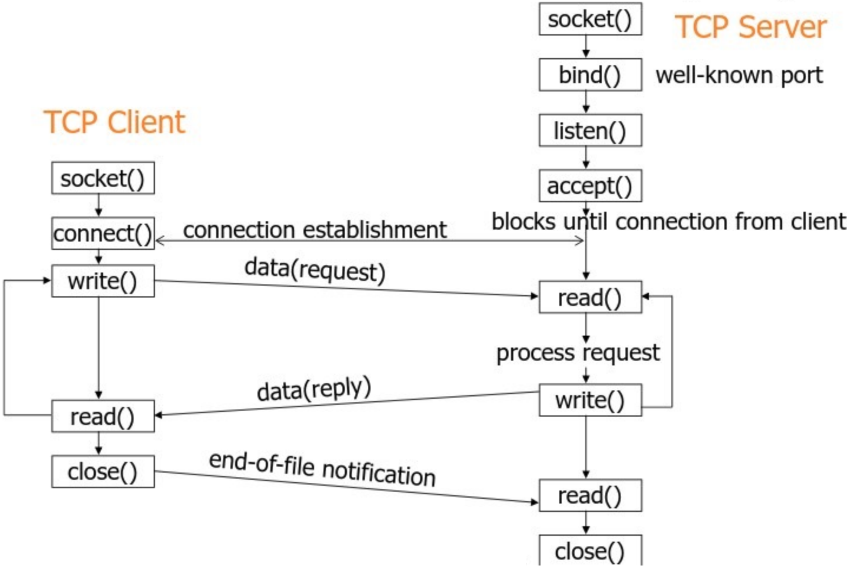
- Worth reading for information in them:
  - sys/socket.h Core socket functions and structures
  - netinet/in.h Protocol families
  - sys/un.h Used for communication within local computer
  - arpa/inet.h Functions to handle numeric IP addresses
  - netdb.h Convert names into numeric IP

Summary

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

// Use TCP for Project 1
sock_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); // TCP
// OR
sock_fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);  // UDP
// OR
sock_fd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);    // IP


//  (or simply 0 for the protocol / third parameter for TCP and UDP)
```

Hello World: Client Side

Client-Server Communication (TCP)



Client: socket()

```c
// For later use
int portno = atoi( argv[2] );     // e.g. 55556
char *server = argv[1];           // e.g. "192.168.1.13"


// socket() is the same on both sides (client/server)

if( (sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    perror("Failed to create socket"); return (-1);
}
```

Client: connect( ) – with inet_pton

```
// Setup socket address structure for connection struct
struct sockaddr_in serv_addr;

memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET ;
serv_addr.sin_port = htons( portno );

// server needs to be a string with the IP address here, e.g. “192.168.1.13"
if( inet_pton(AF_INET, server, &serv_addr.sin_addr) <= 0)
{
  perror(" failed to set socket address");
  exit(0);
}

// Connect to remote address
if( connect( sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
{
    perror(" Could  not connect");
}
```

Client: connect ( ) – with gethostbyname

or getaddrinfo()
(see later)

```
struct sockaddr_in server_addr;
struct hostent *server;

// server can be hostname here, e.g. "skel.ru.is"
server = gethostbyname(server); // map name to host entity

// Fill in fields for server_addr
memset (&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons( portno );

memcpy((char *)&server_addr.sin_addr.s_addr,
       (char *)server->h_addr,
       server->h_length);

// Connect to remote address
connect( sock, (struct sockaddr *)&server_addr, sizeof(server_addr));
```

What is bad about this code?

Client: Sending Hello World!

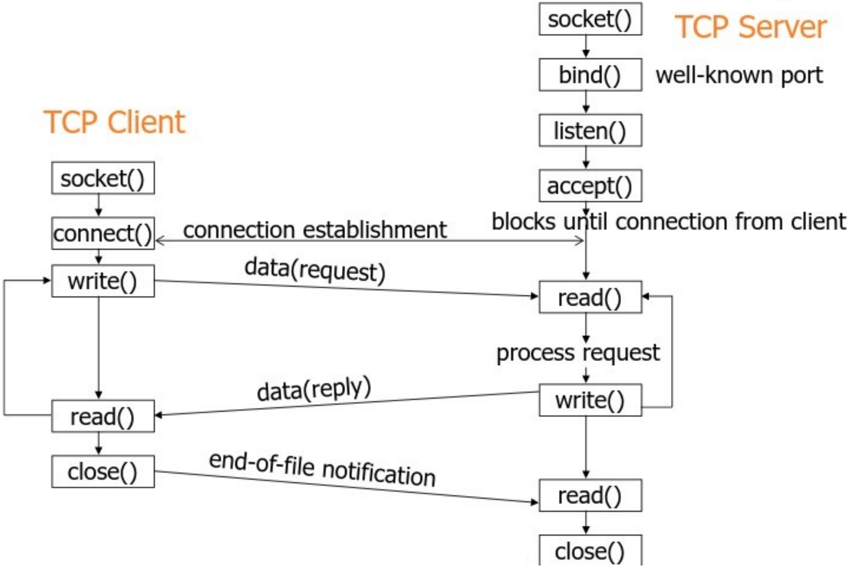
```
// Don’t send the NULL character at end of string

int nsend = send( sock, "Hello World", sizeof("Hello World") – 1, 0);


int nread = read( sock, buffer, sizeof(buffer) );


// … [usually above code is repeated in some loop]
close( sock );
```

* note: sizeof() only works for statically allocated data structures, i.e. size known at compile time. For dynamically allocated strings, use – for example – strlen()

** note2: write(socket,ptr,len) is the same as send(socket, ptr, len, 0)

Hello World: Server Side

Client-Server Communication (TCP)



Server: socket()

```c
// socket() is the same on both sides (client/server)

if( (listenSock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    perror("Failed to create socket"); return (-1);
}
```

Additional Note: Socket Options

```
// Turn on SO_REUSEADDR to allow socket to be quickly reused after
// program exit.
int set = 1;

if( setsockopt( listenSock, SOL_SOCKET, SO_REUSEADDR, &set, sizeof(set)) < 0)
{
    perror("Failed to set SO_REUSEADDR:");
}
```

For list of possible socket options, look at: `man 7 socket`

Server: bind() and listen()

- Next: bind() socket to a port to listen() on
- First: create an address structure to hold the port
  - INADDR_ANY: Bind to all addresses on local host
  - htons(): Convert value from host to network byte order

```
struct sockaddr_in sk_addr;
// Initialise memory
memset (&sk_addr, 0, sizeof(sk_addr));

// Set type of connection
sk_addr.sin_family        = AF_INET;
sk_addr.sin_addr.s_addr   = INADDR_ANY;
sk_addr.sin_port          = htons(portno);

// And bind address/port to socket
if( bind(listenSock, (struct sockaddr *)&sk_addr, sizeof(sk_addr)) < 0)
{
    perror(" Failed  to  bind  to socket:  "); return (-1);
}

listen(listenSock, 5);
```

Notes on byte order

- Network order (for messages etc.) is big-endian
  - i.e. 0x1234 is represented as 0x12, 0x34
- Intel architecture is little-endian
  - i.e. 0x1234 becomes 0x34, 0x12
- htons(), ntohs() and their friends convert to/from network and host order
- Convention is to always use them, even if underlying architecture is also big-endian

Notes on ports

- 16 bit unsigned integer
- Specific to host
- Ports 0 .. 1023 are "well known ports" and are assigned by the OS (applications with system priviledges/root)
- Ports 1024 .. 65535 are available to user applications
- IANA recommendation:
  - 0..1024: system ports
  - 1024 .. 49151: user ports
  - 49152 .. 65535: dynamic/private ports

registered / assigned ports

See https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt

Server: Handling incoming connections

- Clients connect() to the socket specified in listen()
- Servers accept() the connection
  - Then client is handed off to their own two-way **client socket**
  - **listen socket** is specifically used for incoming connections
- Servers then have to
  - Maintain a list of sockets they are communicating with
  - Detect when there is something on those sockets to recv()/read()

Socket sets

```
int listenSock;            // Socket for connections to server
int clientSock;            // Socket of connecting client
fd_set openSockets;        // Current open sockets
fd_set readSockets;        // Socket list for select ()
fd_set exceptSockets;      // Exception socket list
int maxfds;                // Passed to select () as max fd in set

//  Add the listen socket to socket set
{
    FD_ZERO( openSockets );
    FD_SET( listenSock, &openSockets );
    maxfds = listenSock ;   // there is only one socket so far
}

//  Get modifiable copies of openSockets
readSockets = exceptSockets = openSockets;

//  Get a list of sockets waiting to be serviced (blocks while non are waiting)
int n = select( maxfds + 1, &readSockets, NULL, &exceptSockets, NULL );

//  Handle new connections to the server and/or data from some client?
```

Handle new connections to the server

```
if( FD_ISSET ( listenSock, &readSockets ))
{
    struct sockaddr_in client;
    unsigned int clientLen = sizeof(client);
    clientSock = accept( listenSock, (struct sockaddr *)&client,
                         &clientLen );

    // Add new client socket to the list of sockets being monitored
    FD_SET ( clientSock, &openSockets );

    // update the max fd in our socket set
    maxfds = std:: max( maxfds, clientSock );
}
```

Note: Slightly modified version on next slide!

Handle data from some clients

```
int fd;
// Get a list of sockets waiting to be serviced (blocks while non are waiting)
int n = select( maxfds + 1, &readSockets, NULL, &exceptSockets, NULL );
if (n<0) { perror("select failed"); exit(1); }

for (fd=0; fd<=maxfds; fd++) {
   if (FD_ISSET(fd, &readSockets) ) {
      if (fd == listenSock) {
         // code from previous slide for handling new connections
      } else {
         // data from a client
         int nbytes = recv(fd, buffer, sizeof(buffer), 0);
         if(nbytes<=0) { // no data => end of connection / connection error
            close(fd);
            FD_CLR(fd, openSockets);
         } else {
            send_to_all( buffer, nbytes );
         }
      }
   }
}
```

Alternatives to using select()

- Create a process per client
  - fork()

- Create a thread per client
  - pthread_create(…)

If time permits....

Let´s discuss:

What is bad about this code:
https://www.geeksforgeeks.org/tcp-and-udp-server-using-select/

Implications of Real Time

Real Time

- Usually there is some amount of time they must complete all tasks
  - Real time constraint
- Real time programs either:
  - Raise interrupts
  - Operate in a loop
  - Poll
- Examples:
  - Monitoring real time state
  - Receiving a message (e.g. network)

Sequential vs Real Time Debugging

- Sequential:
  - Run the program until it stops, figure it out
  - Interactive debuggers can be used
- Real Time:
  - Interactive debuggers can be very difficult to use
  - Typically use printf() statemens or similar
  - But: adding any code can change timing
    - ”Heisenbugs“
  - Least intrusive: dump logs over UDP to another computer