Operating Systems

Recitation 9

Plan

- Networking background
  - Protocols
  - Layers
  - TCP/IP stack - Ethernet, TCP, DNS etc.
- Sockets & API (Linux)
- Client/Server scheme

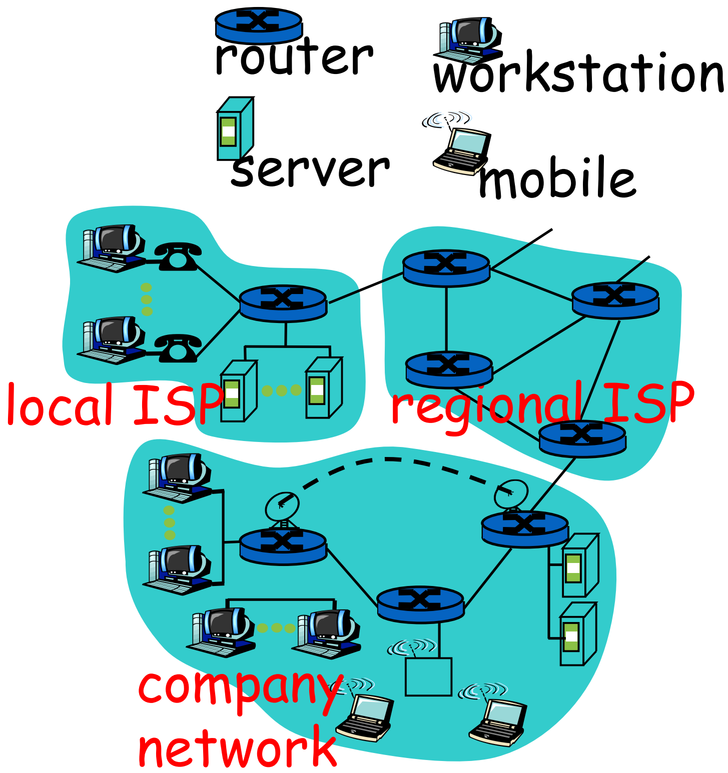
What is the Internet

Internet: "network of networks"

- loosely hierarchical
- public Internet Vs private intranet

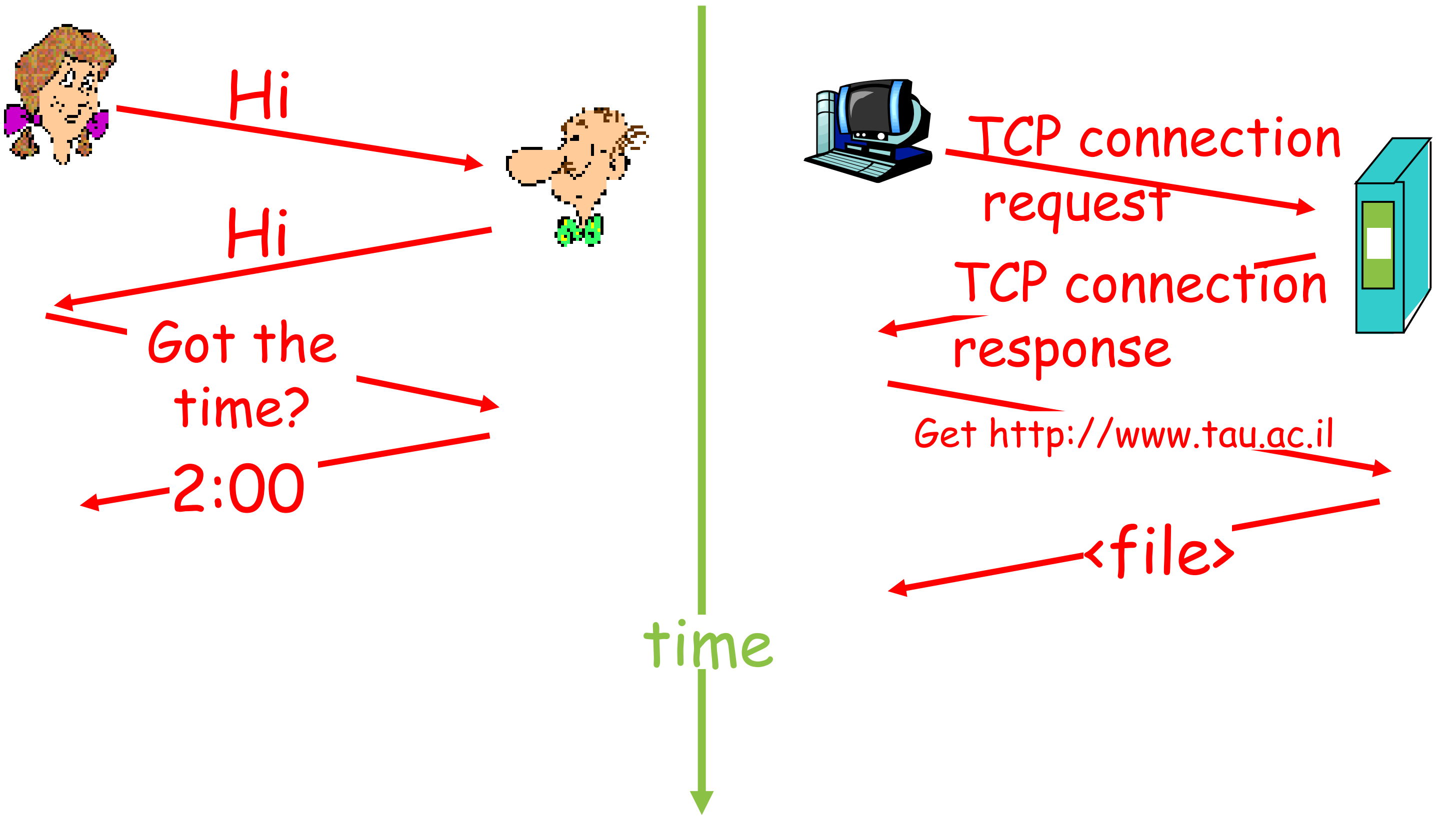
protocols coordinate communication

- Who gets to transmit?
- What path to take?
- What message format?
- e.g., TCP, IP, HTTP, FTP



Protocols

- A human protocol and a computer network protocol:

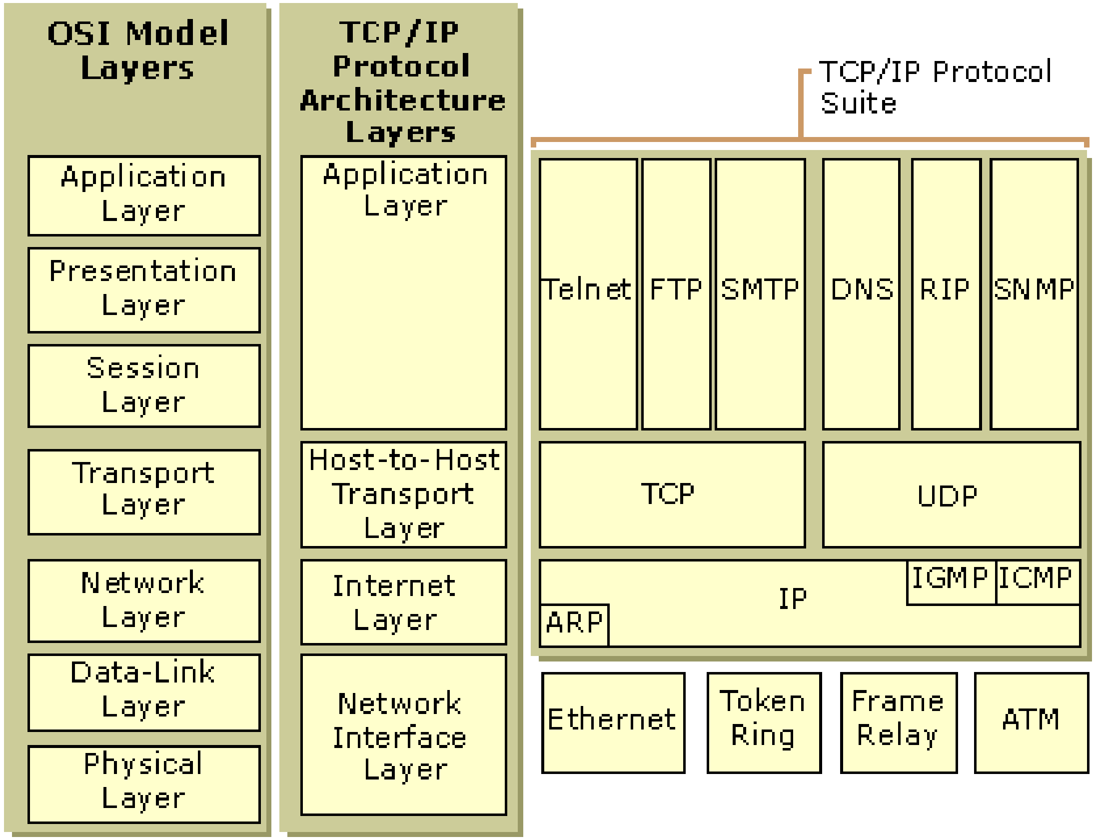


Layering

- Protocols are “stack”-ed in Layers:
- Each layer implements a service
- Same layers on each side of connection communicate data to each other
  - Layers rely on services provided by layer below

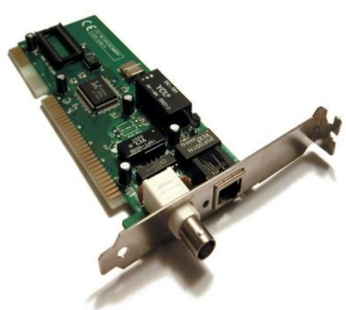
TCP/IP protocol stack

- Most common protocol stack

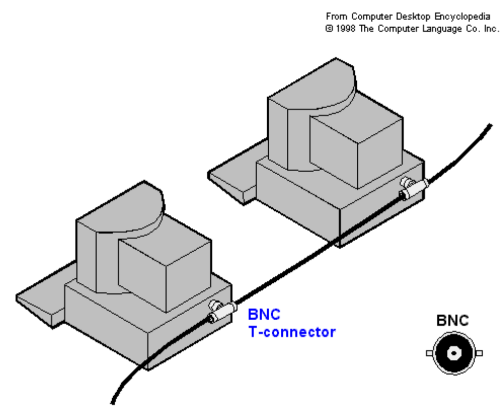


Ethernet: Bus

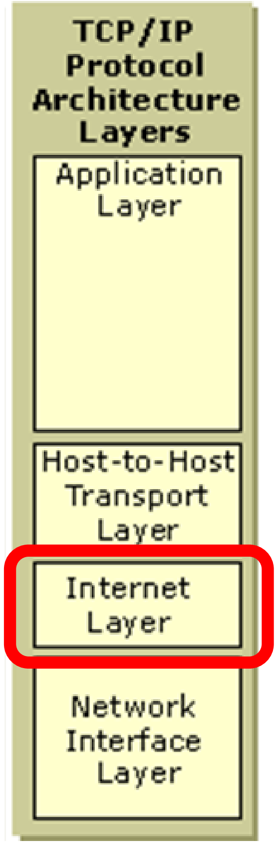
- Network interface layer (lowest)
- Several Stations connected by wire (bus), ("everyone sees everyone")
- Each network card station has unique address
  - MAC == Medium Access Control
- Need to know each other's address on the physical line to communicate data to it
- Nothing is guaranteed



A **network card, network adapter, LAN Adapter** or **NIC**



IP address



- Internet layer
- Numerical address (115.64.32.12- human readable form)
- Global unique addresses to be reached from outside world
- Divided to sub-networks

Subnets

- IP address:
  - subnet part (high order bits)
  - host part (low order bits)
- *What's a subnet ?*
  - device interfaces with same subnet part of IP address
  - can physically reach each other without intervening router
- Local addresses to use within <u>local networks</u> can be <u>reused</u> (10.0.2.X, 192.168.1.X…)

subnet part | host part

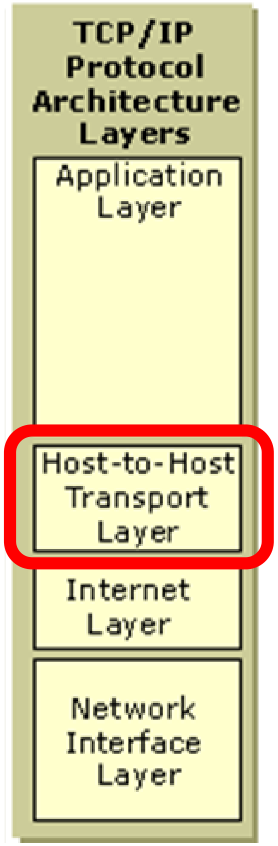
11001000 00010111 00010000 00000000

200.23.16.0/23

IP Routing

- Packets reach their destination **<u>Hop-by-hop</u>**
- Think of how you send a package through the mail from Tel Aviv to Beer Sheva
  - local mail office (TA)
  - regional branch (TA)
  - regional branch (BS)
  - local mail office (BS)
- How do we know where to deliver next (next “hop”)?
  - Use <u>Routing tables</u>

IP is “Best-Offer”



- No guarantees
- Lots of things can go wrong:
  - data corruption
  - lost data packets
  - duplicate arrival
  - out-of-order packet delivery
- To improve, IP packet “data” portion needs to implement some protocol too
  → i.e. transport layer (TCP, UDP)

TCP over IP (TCP/IP)

- **Transmission Control Protocol (TCP)**

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><th>0</th><th>0</th><td colspan="16">Source port</td><td colspan="16">Destination port</td></tr>
<tr><th>4</th><th>32</th><td colspan="32">Sequence number</td></tr>
<tr><th>8</th><th>64</th><td colspan="32">Acknowledgment number (if ACK set)</td></tr>
<tr><th>12</th><th>96</th><td colspan="4">Data offset</td><td colspan="3">Reserved
0 0 0</td><td>N
S</td><td>C
W
R</td><td>E
C
E</td><td>U
R
G</td><td>A
C
K</td><td>P
S
H</td><td>R
S
T</td><td>S
Y
N</td><td>F
I
N</td><td colspan="16">Window Size</td></tr>
<tr><th>16</th><th>128</th><td colspan="16">Checksum</td><td colspan="16">Urgent pointer (if URG set)</td></tr>
<tr><th>20
...</th><th>160
...</th><td colspan="32">Options (if Data Offset > 5, padded at the end with "0" bytes if necessary)
...</td></tr>
</table>

TCP over IP (TCP/IP)

- Guaranteed delivery
  - Packets have sequence number (SQN)
  - Receiver send acknowledge (ACK)
- Port number in addition to IP address
  - To distinguish between applications
- Mechanisms for establishment and termination of a connection

The Importance of Ports

- Ports ==16 bit identifiers
- Uniquely identify processes involved in a socket (TCP & UDP)
- Help route data to destination
- In UNIX first 1024 ports for both protocols are called “well known ports”
- Defined in /etc/services
- Programs that bind to these ports require “root” access

UDP over IP (User Datagram)

Offsets	Octet	0								1								2								3							
Octet	**Bit**	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	**0**	Source port																Destination port															
4	**32**	Length																Checksum															

- Simple transport-layer protocol
- Over IP
- Ports to distinguish applications at the same host
- Length & checksum for verification

UDP over IP (User Datagram)

- No guaranteed delivery
- May arrive out of order (different routing path)
- Light Weight (less delay, processing)

- Usage?
  - Audio streaming
  - Why?

Sockets

- Yet another form of IPC
- Linux/Windows use sockets to communicate between <u>processes</u> over the network
  - remote, but also local!
- Socket is a concept
  - represented in software by its handle
- In Unix its a “file”
  - usually nameless
  - represents a network connection == source/dest IPs & ports + protocol
- Implemented in socket library
  - #include <sys/types.h>
  - #include <sys/socket.h>

socket()

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

- “creates an endpoint for communication and returns a descriptor” - `man 2 socket`
- family - protocol family
  - AF_INET (IPv4 Internet protocols)
  - AF_UNIX (local socket = local IPC!)
- type - communication semantics
  - SOCK_STREAM - full duplex, reliable connection
  - SOCK_DGRAM - datagrams, unreliable, connectionless

socket() cont.

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

- protocol – transport layer protocol to use. 0 chooses default
  - For type=SOCK_STREAM, protocol 0 is TCP
  - What about SOCK_DGRAM?
- Protocols for AF_UNIX? none ☺

- Returns - socket (file) descriptor

Binding a socket

```
int bind(int sockfd, struct sockaddr
*my_addr, int addrlen)
```

- <u>sockfd</u> – returned from socket()
- <u>my_addr</u> – address to bind to, machine IP and desired port
  - Example soon
- <u>addrlen</u> – my_addr length in bytes

- Returns - 0 on success, 1 failure

Listening to a socket

```
int listen(int sockfd, int num)
```

- Mark socket as a “special” - one used to accept incoming connection requests
  - SOCK_DGRAM not legal. Why?
- sockfd – the socket
- num – max # of pending connections

- Returns – 0 success, 1 failure

Accepting a Connection

```
int accept(int sockfd, const struct sockaddr*
           serv_addr, socklen_t addrlen)
```

- Wait on socket for incoming requests.
  - SOCK_DGRAM not legal…
- Create new socket with new descriptor
- <u>sockfd</u> – listening socket
- <u>serv_addr</u> – *remote* server address
  - Example soon
- <u>addrlen</u> - # of bytes in serv_addr
- Returns – new sockfd on success, -1 failure

Connecting to a Socket

```
int connect(int sockfd, const struct sockaddr*
            serv_addr, socklen_t addrlen)
```

- sockfd – socket
- serv_addr – *remote* server address
  - Example soon
- addrlen - # of bytes in serv_addr

- Returns – 0 success, 1 failure

Address data structures

- Format and size of my_addr is usually protocol specific.
- struct sockaddr is used as the “base” of a set of address structures

```
struct sockaddr {
  unsigned short sa_family; // address family, AF_xxx
  char sa_data[14]; // 14 bytes of protocol address
};
```

Address data structures (2)

```
struct sockaddr_in {
    short int sin_family; // address family, AF_xxx
    unsigned short int sin_port; // port number
    struct in_addr sin_addr; // internet address
    unsigned char sin_zero[8]; // for alignments
};
struct in_addr {
    unsigned long s_addr; //32-bit long (4 bytes) IP
                                                address
}
```

Typical Usage

```
struct sockaddr_in serverName;
serverName.sin_family = AF_INET;
…
connect(clientSocket,
              (struct sockaddr*) &serverName,
               sizeof(serverName));
```

Communicating data

```
int send(int sockfd,const void *msg,
          size_t len, int flags)
int recv(int sockfd, void *buf,
          size_t len, int flags)
```

- Send data, with special options and flags.

- Conveniently can also be done like regular files with **`read()/write()`** on `sockfd`
- By default <u>blocking</u> = control returned to process only when write/read occurs
  - Though its configurable

read()/write() semantics

- `read()`
  - blocks until data is available
  - may return < than max bytes (whatever is available)
  - you must be prepared to read data 1 byte at a time!
  - **How do we know when to finish?**

- `write()`
  - might not be able to write all bytes
  - returns # of bytes sent (not received…)

Close socket

```
int close(int sockfd)
```

- Close socket descriptor pointed by `sockfd`
- Either end of the connection can `close()`
- If other end closed connection, and there is no buffered data, reading from a TCP socket returns 0 to indicate EOF.

- Returns – 0 success, 1 failure

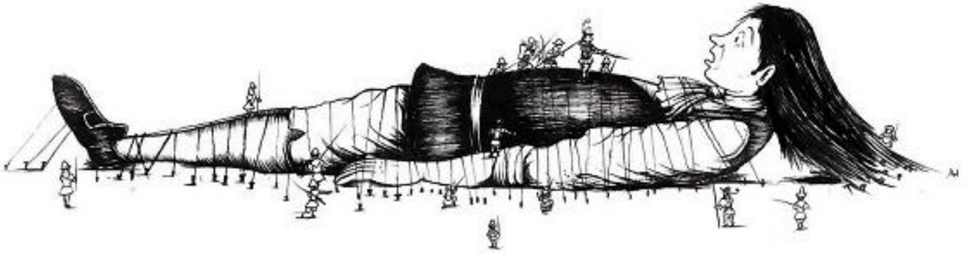
client	server
sd=socket()	sd=socket()
	bind(sd, port)
	listen(sd,…)
connect(sd, dst)	new_sd=accept(sd)
write(sd, …)	write(new_sd, …)
read(sd, …)	read(new_sd, …)
close(sd)	close(new_sd)

Some explaining regarding client/server

- Why do we need `bind()`?
  - Otherwise server listens on **random** port. Clients don’t know where to connect!
- Why do we need listen?
  - Possibly multiple clients send requests
  - Listen defines length of queue for such incoming connections
- Main thread does all the work here!
  - May want to fork or delegate work to threads, so main thread can continue servicing incoming connections

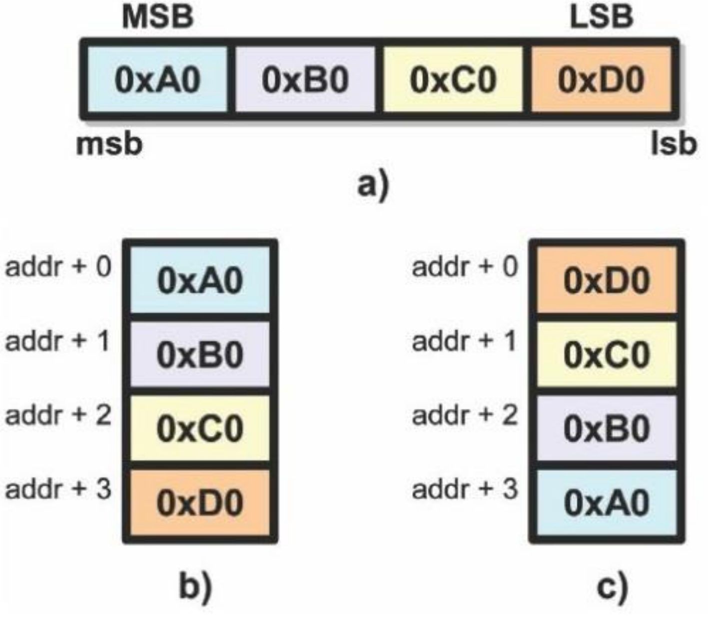
Endianness

- One last note before code
- A long long time ago in Lilliput, society split into two factions!
  - Big-Endians - opened soft-boiled eggs at larger end ("the primitive way")
  - Little-Endians - broke their eggs at the smaller end.
- The feud continues…



Endianness

- Different CPUs have different order of bytes – MSB, LSB
  - **Big-Endian** - MSB of any multibyte data field is stored at the lowest memory address which is also the address of the larger field.
  - **Little-Endian** – same but for LSB



Endianness

- Why do we care? we communicate between remote computers, with possibly different CPUs
  - X86 little endian
- When we pass numbers, need uniform representation

- Socket library utilities

```
u_long htonl(u_long); // host to network long (32 bits)
u_short htons(u_short); // host to network short (16 bits)

u_long ntohl(u_long); // network to host long (32 bits)
u_short ntohs(u_short); // network to host short (16 bits)
```

Example

- TCP server/client
- ***Code example***