Communication Networks (0368-3030) / Spring 2011

The Blavatnik School of Computer Science, Tel-Aviv University

Allon Wagner

Staff

- Lecturer: Dr. Eliezer Dor
  - eliezer.dor @ gmail
  - Office hours: by appointment
- Teaching Assistant: Allon Wagner
  - allonwag @ post
  - Office hours: Mon. 12-13 Orenstein 410, or by appointment
- HW Grader:
  - Michael Shifman
  - shifman@mail.tau.ac.il

Homework

- 3 practical assignments
  - “hands-on” network programming
  - C / C++
- 4-5 theoretical assignments
  - will probably include some guided-reading – bonus points
  - Guided-reading is considered part of the material for the final exam
- Moodle forum for HW related questions

Requirements & Grading

- Final Exam 60%
- Practical HW assignments 20%
- Theoretical HW assignments 20%

- Submission of all the assignments is mandatory
- HW may be submitted in pairs
- There will be a closed-books final exam
  - You may bring 4 pages (i.e. 2 two-sided sheets) with you to the exam

Textbooks & Online Material

- Course website: http://www.cs.tau.ac.il/~allonwag/comnet2011B/index.html
- Main textbook:
  - Computer Networking: A Top-down Approach, by J. F. Kurose and K. W. Ross (3rd edition or later).
- Other references:
  - Computer Networks, by A. S. Tanenbaum (4th edition or later).
  - Computer Networks: A Systems Approach, by L. L. Peterson and B. S. Davie (3rd edition or later).
  - An Engineering Approach to Computer Networking, by S. Keshav.
- Wikipedia, and lots of online material

Why study computer networks?

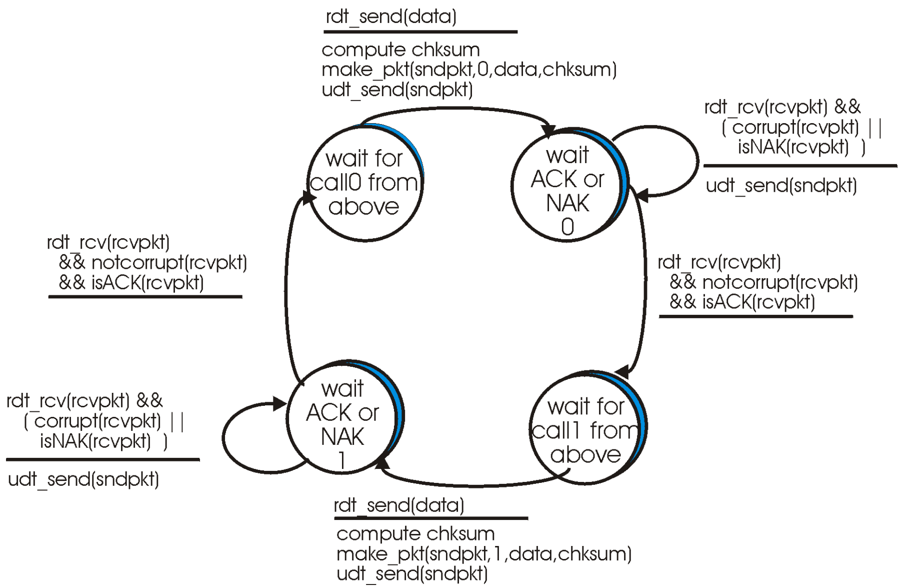
- An interface between theory (algorithms, mathematics) and practice
- Understanding the design principles of a truly complex system
- Industry-relevant knowledge
- Fun!

- Challenges in teaching computer networks
- Students' feedback

Introduction

Protocols

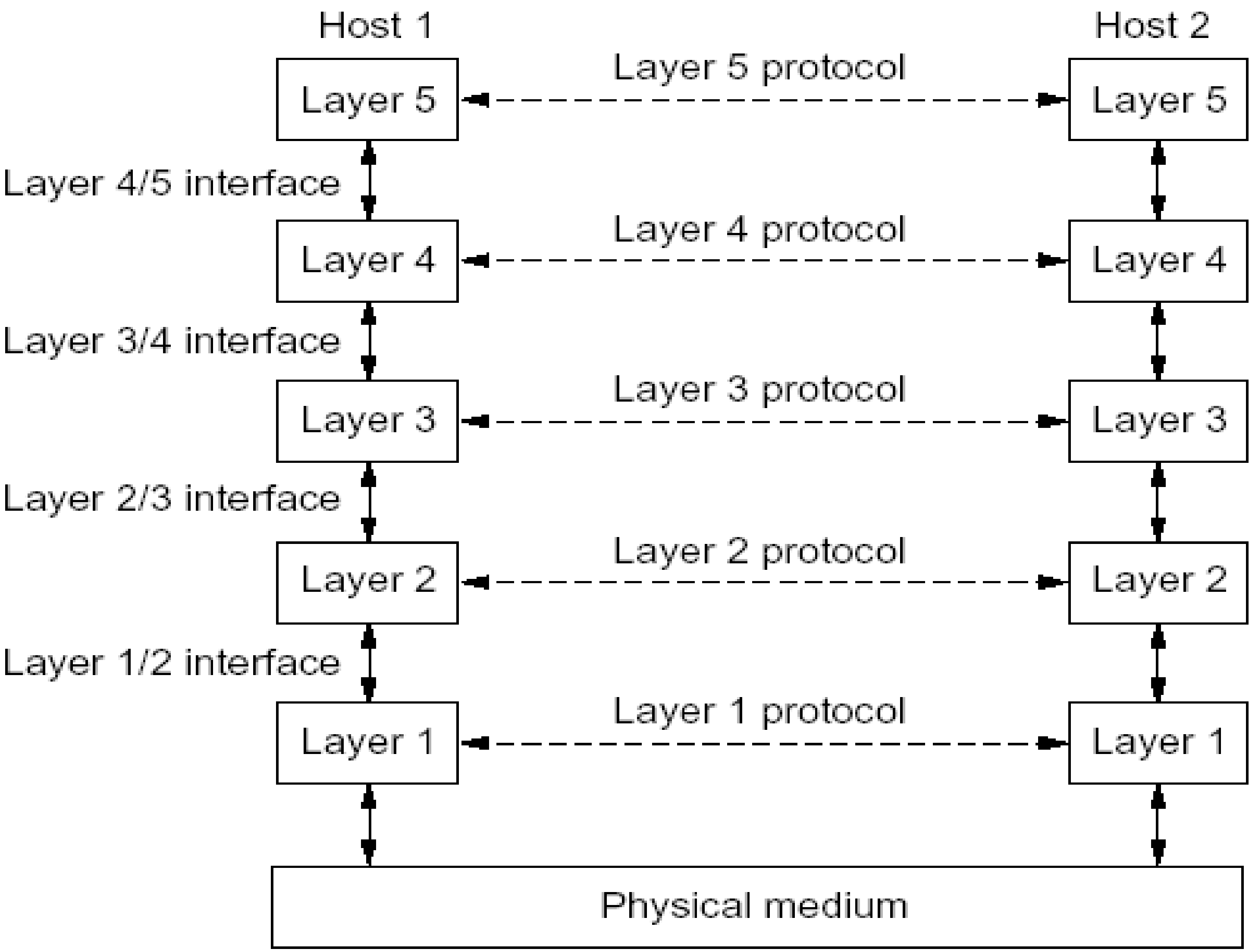
- A protocol defines:
  - Format (Syntax)
  - Conversation logic
  - → Finite state machine!
- Open/ proprietary



Networking is a complex task

- Solution: modularity
  - Layering
  - Transparency
  - Each layer is dependent only on the interfaces defined by the layers above and below it
  - Each layer “talks” only to its equivalent on the remote side
  - Each layer is implemented by a protocol

Layering



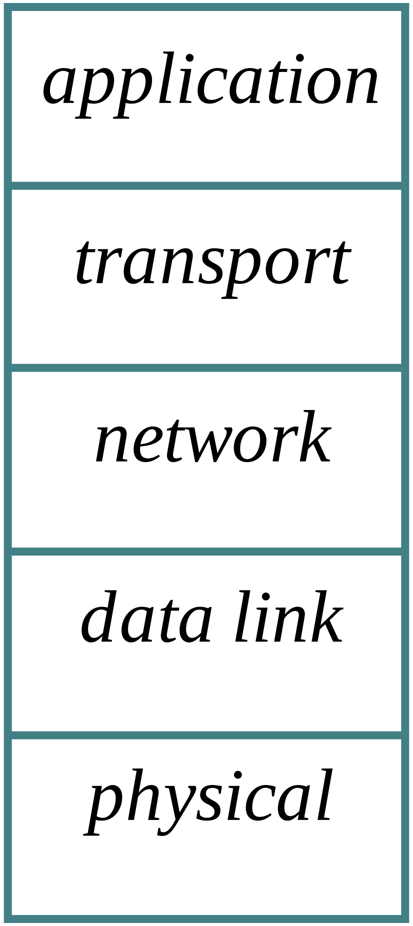
Host 1
Host 2
Layer 5 protocol
Layer 5
Layer 5
Layer 4/5 interface
Layer 4 protocol
Layer 4
Layer 4
Layer 3/4 interface
Layer 3 protocol
Layer 3
Layer 3
Layer 2/3 interface
Layer 2 protocol
Layer 2
Layer 2
Layer 1/2 interface
Layer 1 protocol
Layer 1
Layer 1
Physical medium

Layering Models

- OSI Reference Model
  - 7 layers
  - Defined by ISO (International Standards Organization)
  - Widely used as a reference model, but seldom implemented
- TCP/IP Reference Model
  - 5 layers
  - Protocols came first, the model is actually a description of their workings.
  - The TCP/IP suite is the backbone of today's Internet.

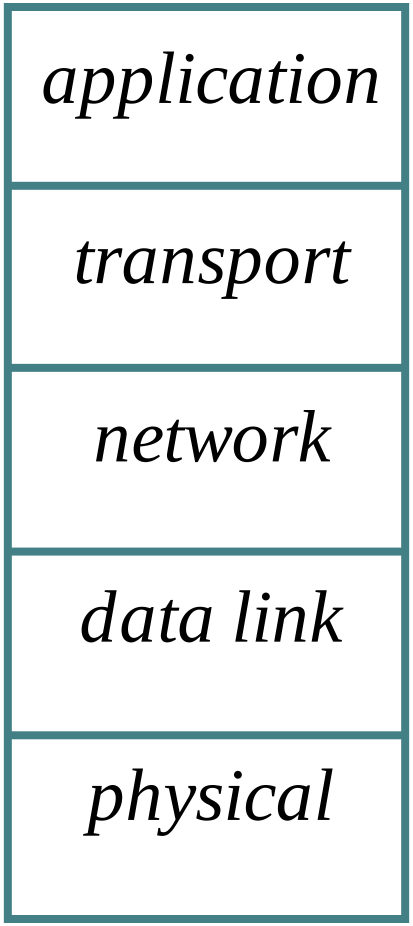
Overview of the 5-layers model

- Physical layer
  - Transmits raw bits over a communication channel
- Data link layer
  - Control layer over the physical layer
  - Framing
- Network layer
  - Delivers packets from source to destination across the network
  - Routing vs. Forwarding
  - In TCP/IP: IP is the forwarding protocol

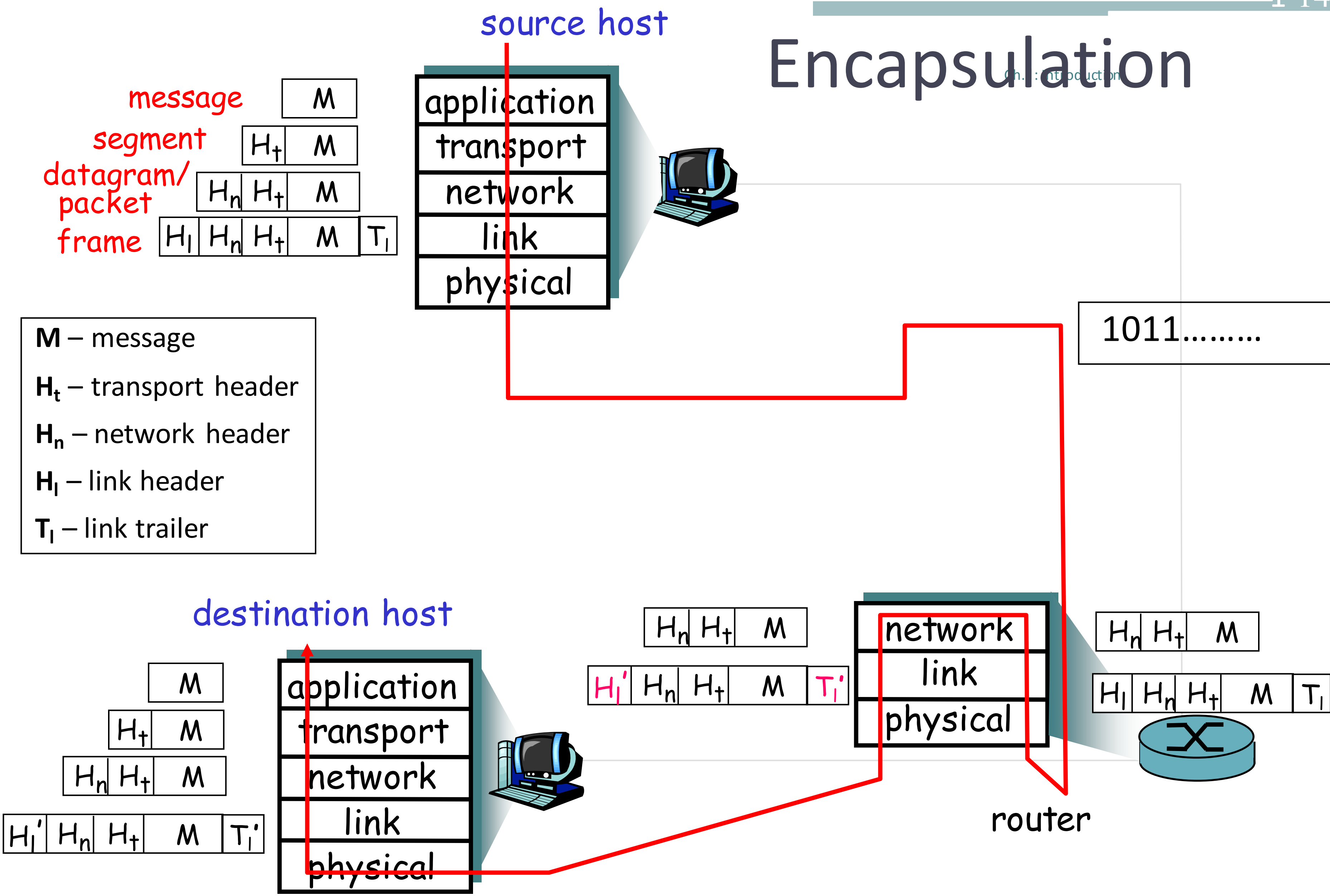


Overview of the 5-layers model (cont.)

- Transport layer
  - Delivers data between a program on the source machine to a peer program on the host machine.
  - First end-to-end layer!
  - In TCP/IP:
    - TCP: reliable, connection-oriented
    - UDP: unreliable, connectionless
- Application layer
  - A protocol (sometimes a protocol stack) to implement the desired application service.
  - Examples:
    - Mail: SMTP, POP3, IMAP
    - Remote control: Telnet
    - File transfer and sharing: FTP, Bittorrent
    - Instant messaging: XMPP (Jabber)



Encapsulation



HW Objective: Write a network application

- Design an application protocol
  - Syntax
  - Semantics
  - Conversation logic
- Implement via socket programming
  - An interface to the OS's transport layer

Socket Programming – Part I

Recommended References:

Beej's Guide to Network Programming

http://beej.us/guide/bgnet/

Unix Network Programming \ W. Richard Steven

Slides for this topic, as well as other topics along the course, are partly based on the work of previous teaching assistants to this course: Hillel Avni, Yahav Nussbaum, David Raz, Motti Sorani, Alex Kesselman.

IP Address / Domain Names

- “Uniquely” identifies a “host” on the network
  - Not really, we’ll get to that later in the course
- A 32-bit number
  - For convenience represented as 4 numbers in the range 0-255
  - e.g. 132.67.192.133
- Domain names
  - 132.67.192.133 = nova.cs.tau.ac.il

Port

- A 16-bit number (i.e., 0-65535)
- Identifies a service on the host
  - Again, not quite, we'll get to that later, blah-blah.
  - For instance: HTTP = 80, SMTP = 25, Telnet = 23

- A socket is a combination of IP + port
  - 132.67.192.133 : 80

Port (cont.)

- The server listens on a certain port
- The client randomly chooses a port to which the server answers

- For instance
  94.127.73.5 : 1902 ↔132.67.192.133 : 80

Relevant Headers

- #include <sys/socket.h>
  - Sockets
- #include <netinet/in.h>
  - Internet addresses
- #include <arpa/inet.h>
  - Working with Internet addresses
- #include <netdb.h>
  - Domain Name Service (DNS)
- #include <errno.h>
  - Working with errno to report errors

Address Representation

```
struct sockaddr {
  u_short sa_family;
  char sa_data[14];
};
```

- sa_family
  - specifies which address family is being used
  - determines how the remaining 14 bytes are used

Address Representation – Internet Specific

```
struct sockaddr_in {
  short sin_family; /* = AF_INET */
  u_short sin_port;
  struct in_addr sin_addr;
  char sin_zero[8]; /* unused */
};

struct in_addr {
 uint32_t     s_addr;
}
```

- Except for sin_family, all contents are in **network order**

Big Endian / Little Endian

- Memory representation of multi-byte numbers:
  - $2882400018_{10} = ABCDEF12_{16}$
  - Big Endian: 0xAB CD EF 12
  - Little Endian: 0x 12 EF CD AB
- Hosts on the web use both orders
- On the network all use big endian (= network order).
- Numbers used for port number, IP etc. should thus be converted
  - htonl () / ntohl() / htons() / ntohs()

Reliable vs. Unreliable Sockets

SOCK_STREAM	SOCK_DGRAM
reliable transport	unreliable transport
connection-oriented	connectionless
keeps state	stateless
more resources needed	lightweight
TCP	UDP

Session overview

- We will start with reliable transport (TCP)

Client	TCP	Server
		socket()
		bind()
socket()		listen()
connect()	← session setup →	accept()
send()	data transfer →	recv()
recv()	← data transfer	send()
close()	←terminate session→	close()

Socket Creation – socket()

- `int` **`socket`**`(int` *`domain`*`, int` *`type`*`, int` *`protocol`*`);`
- domain: PF_INET for IPv4
- type: for our purposes either **`SOCK_STREAM or`** SOCK_DGRAM
- protocol: can be set to 0 (default protocol)
- Returns the new socket descriptor to be used in subsequent calls, or -1 on error (and errno is set accordingly).
- Don't forget to close the socket when you're done with it

Bind socket to IP and port – bind()

- `int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);`
- sockfd : socket descriptor
- my_addr: address to associate with the socket
  - The IP portion often set to INADDR_ANY which means "local host"
- addrlen: set to sizeof(my_addr)
- Returns 0 on success, or -1 on error (and errno is set accordingly).

Wait for an incoming call – listen()

- `int` **`listen`**`(int` *`sockfd`*`, int` *`backlog`*`);`
- sockfd : socket descriptor
- backlog: number of pending clients allowed, before starting to refuse connections.
- Returns 0 on success, or -1 on error (and errno is set accordingly).

Accept an incoming connection – accept()

- `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`
- sockfd : socket descriptor
- addr: filled in with the address of the site that's connecting to you.
- addrlen: filled in with the sizeof() the structure returned in the *addr* parameter
- Returns the newly connected socket descriptor, or -1 on error, with *errno* set appropriately.
- Don't forget to close the returned socket when you're done with it

Server-side example

```
sock = socket(PF_INET, SOCK_STREAM, 0);

myaddr.sin_family = AF_INET;
myaddr.sin_port = htons( 80 );
myaddr.sin_addr = htonl( INADDR_ANY );

bind(sock, &myaddr, sizeof(myaddr));

listen(sock, 5);

sin_size = sizeof(struct sockaddr_in);
new_sock = accept(sock, (struct sockaddr*)
  &their_addr, &sin_size);
```

- In real-life code, don’t forget to check for errors

Session overview

- Reliable transport (TCP)

Client	TCP	Server
		socket()
		bind()
socket()		listen()
connect()	← session setup →	accept()
send()	data transfer →	recv()
recv()	← data transfer	send()
close()	←terminate session→	close()

Connect to a listening socket – connect()

- `int` **`connect`**`(int` *`sockfd`*`, const struct sockaddr *`*`serv_addr`*`, socklen_t` *`addrlen`*`);`
- sockfd : socket descriptor
- serv_addr: the address you're connecting to.
- addrlen: filled with sizeof(serv_addr)
- Returns 0 on success, or -1 on error (and errno is set accordingly).

- Most of the times, no bind() is required on the client side:
  - If bind() wasn't called, the local IP address and a random high port are used.

Client-side example

```
sock = socket(PF_INET, SOCK_STREAM, 0);

dest_addr.sin_family = AF_INET;
dest_addr.sin_port = htons( 80 );
dest_addr.sin_addr = htonl(0x8443FC64);

connect(sock, (struct sockaddr*)
  &dest_addr, sizeof(struct sockaddr));
```

- In real-life, the server’s IP is not hard-coded
- In real-life code, don’t forget to check for errors

Session overview

- Once the session is initiated, both parties are equal:
  - Both can send and receive data
  - Both can decide it's time to close the connection
- As long as the listening socket is open, it can accept new incoming clients
  - by calling accept()

Active	Passive
socket()	socket()
…	bind()
…	listen()
connect()	accept()
Connected	
close()	close()
…	…
…	accept()

Closing a connection – close()

- `int close(int sockfd);`
- sockfd : socket descriptor
- returns 0 on success, or -1 on error (and errno is set accordingly)

- After we close a socket:
  - If the remote side calls **recv()**, it will return 0.
  - If the remote side calls **send()**, it will receive a signal SIGPIPE and send() will return -1 and errno will be set to EPIPE.
- shutdown() can be used to close only one side of the session
  - Rarely used
  - Refer to the man pages

Session overview

- Unreliable transport (UDP)

Client	UDP	Server
		socket()
socket()		bind()
sendto()	data transfer →	recvfrom()
recvfrom()	← data transfer	sendto()
close()		close()

Sending data (TCP + UDP)

- TCP: `ssize_t` **`send`**`(int` *`socket`*`, const void *`*`buffer`*`, size_t` *`length`*`, int` *`flags`*`);`
- UDP: `ssize_t` **`sendto`**`(int` *`socket`*`, const void *`*`buffer`*`, size_t` *`length`*`, int` *`flags`*`, const struct sockaddr *`*`dest_addr`*`, socklen_t` *`dest_len`*`);`
- buffer, length: buffer of the data to send, and number of bytes to send from it.
- flags: send options. Refer to the man pages. Use 0 for “no options”.
- In unconnected sockets (UDP) you specify the destination in each sendto().

Partial send

- send() and sendto() return the number of bytes actually sent, or -1 on error (and errno is set accordingly).
- The number of bytes actually sent might be less than the number you asked it to send.

A code considering that

(Use it for TCP. For UDP it makes less sense – we will discuss later)

```
int sendall(int s, char *buf, int *len) {
    int total = 0;        // how many bytes we've sent
    int bytesleft = *len; // how many we have left to send
    int n;

    while(total < *len) {
        n = send(s, buf+total, bytesleft, 0);
        if (n == -1) { break; }
        total += n;
        bytesleft -= n;
    }
    *len = total; // return number actually sent here
    return n == -1 ? -1:0; //-1 on failure, 0 on success
}
```

Source: **Beej's Guide to Network Programming**

Receiving data (TCP + UDP)

- TCP: *ssize_t* **recv**(*int socket, void *buffer, size_t length, int flags*);
- UDP: *ssize_t* **recvfrom**(*int socket, void *buffer, size_t length, int flags, struct sockaddr *from_addr, socklen_t from_len*);
- buffer, length: allocated space for the received data, and its size (= max data received by this call)
- flags: receive options. Refer to the man pages. Use 0 for “no options”.

Receiving data (TCP + UDP) (cont.)

- recv() and recvfrom() return the number of bytes received, or -1 if an error occurred (and errno is set accordingly).
- In TCP sockets, 0 is returned if the remote host has closed its connection.
  - This is often used to determine if the remote side has closed the connection.
- In unconnected sockets (UDP) *from_addr* will hold upon return the source address of the received message.
- *from_len* should be initialized before the call to sizeof(*from_addr)*. It is modified on return to indicate the actual size of the address stored in *from_addr*.

Translating a host name to an IP address

- `struct hostent *`**`gethostbyname`**`(const char *`*`name`*`);`
  - deprecated
- `int` **`getaddrinfo`**`(const char *`*`hostname`*`, const char *`*`servname`*`, const struct` *`addrinfo *hints`*`, struct` *`addrinfo **res`*`);`
- Supports many options and thus seems complex, but basic use is simple.
  - Refer to Beej's guide for more info and for a simple example of its use: http://beej.us/guide/bgnet/output/html/multipage/getaddrinfoman.html
- Don't forget to use *freeaddrinfo*() to release memory when you're done with *getaddrinfo*'s result.

Other Useful Functions

- `inet_ntop(), inet_pton()`
  - Convert IP addresses to human-readable text and back
- `getpeername()`
  - Return address info about the remote side of the connection.
  - Used after calling accept() (server) or connect() (client)
- `gethostname()`
  - returns the standard host name for the current processor

What do we send?

Tips for defining a protocol

Binary protocols

- Uniform endianity for numbers
- String representation:
  - Bad: decide on maximal length
    hello =
    0x 68 65 6C 6C 6F 00 00 00 00
  - Better: use a length field
    hello =
    0x 05 00 68 65 6C 6C 6F
    (note that the integer is in little endian)
- Length field can also be applied to fields of variable length (e.g., options)

An example:

- A DNS response for the query www .icann.org:

  91 73 81 80 00 01 00 01 00 00
  00 00 03 77 77 77 05 69 63 61
  6e 6e 03 6f 72 67 00 00 01 00
  01 c0 0c 00 01 00 01 00 00 02
  58 00 04 c0 00 20 07
- For instance, bytes 0-1 are transaction ID, bytes 2-3 hold various flags.
- Text view:

  .s..........www
  .icann.org............. X.... .

Textual Protocols – An example

HTTP request for the page
http://www.ietf.org/rfc/rfc3514.txt

GET /rfc/rfc3514.txt HTTP/1.1

Host: www.ietf.org

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7

Keep-Alive: 115

Connection: keep-alive

The response:

HTTP/1.1 200 OK

Date: Sun, 13 Feb 2011 14:32:45 GMT

Last-Modified: Fri, 28 Mar 2003 18:36:14 GMT

Content-Encoding: gzip

Content-Length: 4486

Keep-Alive: timeout=15, max=100

Connection: Keep-Alive

Content-Type: text/plain

Know the difference between TCP and UDP

TCP

- Reliable
- Transfers a *stream* of data
  - send() and recv() do not necessarily match message boundaries!
  - Can receive multiple messages together / parts of messages.
  - The application protocol must define a way to separate messages within the stream.
- Affected by congestion – avoidance mechanism etc.

UDP

- Unreliable
  - Should consider that when working with UDP
  - e.g., set a timeout when sending a query and waiting for a response
- Transfers *datagrams*

Word of caution - packing

- Assume you want to have a struct represent your protocol header (or part of it)

```
struct ProtocolHeader {
 unsigned short datagramLength;
 unsigned short datagramType;
 unsigned char flag;
 //...
};
```

Word of caution – packing (cont.)

- Compiler may add padding to guarantee alignment
  - Simply sending the struct "as-is" is not portable
- Output:
  - `0 4 8 16`
  - `S's size is: 24`

```
#include <stdio.h>
#include <stddef.h>

struct S {
    short i;        //2 bytes
    int j;          //4 bytes
    char k;         //1 byte
    double l;       //8 bytes
};

int main()
{
    printf("%ld ", offsetof(S, i));
    printf("%ld ", offsetof(S, j));
    printf("%ld ", offsetof(S, k));
    printf("%ld\r\n", offsetof(S, l));
    printf("S's size is: %ld\r\n\r\n", sizeof(S) );
}
```

Word of caution – packing (cont.)

- Possible solution: use #pragma pack and #pragma pop
  - Code portability issues
- Output:
  - `0 2 6 7`
  - `T's size is: 15`

```
#include <stdio.h>
#include <stddef.h>

#pragma pack(push, 1)
struct T {
    short i; //2 bytes
    int j;          //4 bytes
    char k;         //1 byte
    double l;       //8 bytes
};
#pragma pack(pop)

int main()
{
    printf("%ld ", offsetof(T, i));
    printf("%ld ", offsetof(T, j));
    printf("%ld ", offsetof(T, k));
    printf("%ld\r\n", offsetof(T, l));
    printf("T's size is: %ld\r\n\r\n", sizeof(T) );
}
```

Socket Programming – Part II

Handling blocking calls

Blocking function calls

- Many of the functions we saw block until a certain event
  - accept: until a client initiates a session
  - connect: until the connection is (half) established
  - recv, recvfrom: until a data is received
  - send, sendto: until data is pushed into the socket's buffer
- For simple programs, blocking is convenient
- What about more complex programs?
  - multiple connections
  - simultaneous sends and receives
  - simultaneously doing non-networking processing

How do we handle blocking?

- Initiate multiple threads
- Do not allow blocking by the use of fcntl()
- Call a function only when it's guaranteed not to block
  - select(), pselect(), poll(), ppoll()
  - select() gets a set of fd's and returns which of them is
    - Read-ready: recv() (data socket) or accept() (listening socket) will not block
    - Write-ready: send() will not block

select()

- `int` **`select`**`(int` *`nfds`*`, fd_set *`*`readfds`*`, fd_set *`*`writefds`*`, fd_set *`*`exceptfds`*`, struct timeval *`*`timeout`*`);`
- nfds: highest-numbered file descriptor in any of the three sets, plus 1.
- *readfds, writefds, exceptfds: sets of fd*'s to see if they're read-ready, write-ready or except-ready
  - "Exceptional conditions" are <u>not</u> errors, but rather states of the sockets (e.g. TCP's urgent ptr is set).
  - Any set can be replaced with NULL → the corresponding condition will not be checked.

select() (cont.)

- Returns when at least one of the watched fd's becomes ready, or when the timeout expires
  - Returns the total number of ready fd's in all the sets. The sets are changed to indicate which fd's are ready.
  - Returns 0 if timeout expired
  - Returns -1 on error (and errno is set accordingly).

Working with fd_set

- fd_set is just a bit vector
- `void` **`FD_ZERO`** *`(fd_set *set)`*
  - Initializes to an empty set
- `void` **`FD_SET`** *`(int fd, fd_set *set)`*
  - Adds fd to the set
- `int` **`FD_ISSET`** *`(int fd, fd_set *set)`*
  - Returns non-zero value if fd is in the set, 0 otherwise
- `void` **`FD_CLR`** *`(int fd, fd_set *set)`*
  - Removes fd from the set
- stdin, stdout, stderr are associated with fd's 0, 1, 2 respectively

select‘s timeout argument

```
struct timeval {
  long tv_sec; /* seconds */
  long tv_usec; /* microseconds, always less
  than 10^6 */
};
```

- Pass (0,0) to return immediately
- Pass NULL pointer to wait indefinitely until one of the fd’s is ready
- Some OS’s decrease the time elapsed, some don’t
  - Linux does

select example:
reading from multiple active sockets

```
fd_set read_fds;

// main loop of the program
for(;;) {
    FD_ZERO(&read_fds); //reset fd set
    FD_SET(listening_sock, &read_fds);
    for(/* for each active client with fd = client_sock */) {
         FD_SET(client_sock, &read_fds);
    }
    fdmax = //… the highest fd in read_fds

    select(fdmax + 1, &read_fds, NULL, NULL, NULL);

    if (FD_ISSET(listening_sock , &read_fds)) {
         // listening socket is read-ready: a new client is available.
         // new_client_sock = accept(listening_sock, …
    }

    for(/* for each active client with fd = client_sock */) {
         if (FD_ISSET(client_sock , &read_fds)) {
              // client socket is read ready - unread data is available
              // nbytes = recv(client_sock, …
    }
} //END main program loop
```