Operating System Design

Computer Networks in 10 minutes!
Network Programming

Neda Nasiriani
Fall 2018

Computer Network

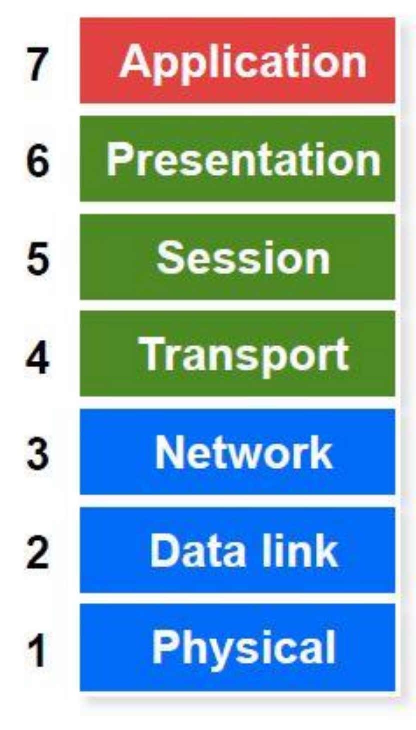
- How does two computers communicate?
- Internet is connecting millions of users all over the globe, who are connecting using different devices
- How does it work?!?
- The key to Internet success is its abstract design and protocols based on those design paradigms (guidelines)

Computer Network

- This architecture which is a layered architecture allows for communication between different nodes as long as they follow the same protocols
  - Remember your last lab on pipes
  - Pipes are an implementation of message passing
  - How did you send an integer and string using the pipe?
- The idea of an abstraction
  - To have a unifying model
  - To encapsulate this model in an object which provides an interface for other layers
  - To hide the details of how the object is implemented from the users of the object.

Open Systems Interconnection (OSI) Model

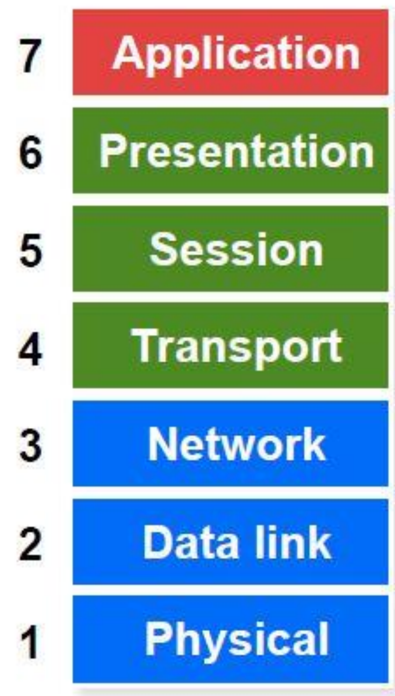
- OSI model is the standard proposed for computer networks
- Partitions the network functionality into seven layers
- Reference model for a protocol
- Physical layer?
  - handles the transmission of *raw bits* over a communications link (wireless, fiber, coax)
- The data link layer?
  - collects a stream of bits into a larger aggregate called a *frame*.
  - Network adaptors, along with device drivers running in the node's OS, typically implement the data link level.



Hui Zhang, 15-441 Networking, Fall 2007, School of computer science, CMU.

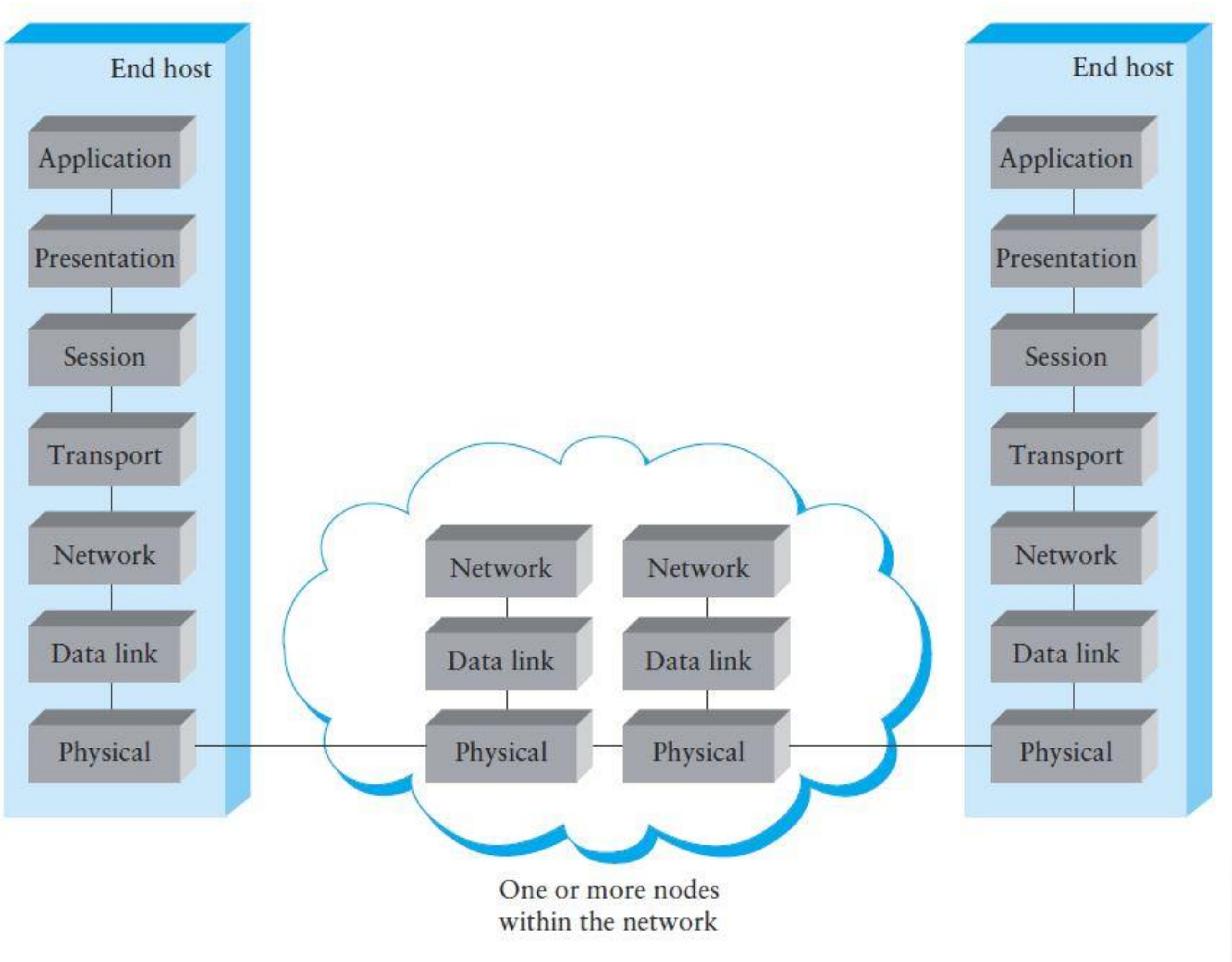
Open Systems Interconnection (OSI) Model

- The network layer?
  - handles routing *packets* among nodes within a packet-switched network.
- The **Transport** layer then implements a process-to-process channel.
  - Here, the unit of data exchanged is commonly called a *message* rather than a packet or a frame.



Hui Zhang, 15-441 Networking, Fall 2007, School of computer science, CMU.

OSI Model on Different Nodes

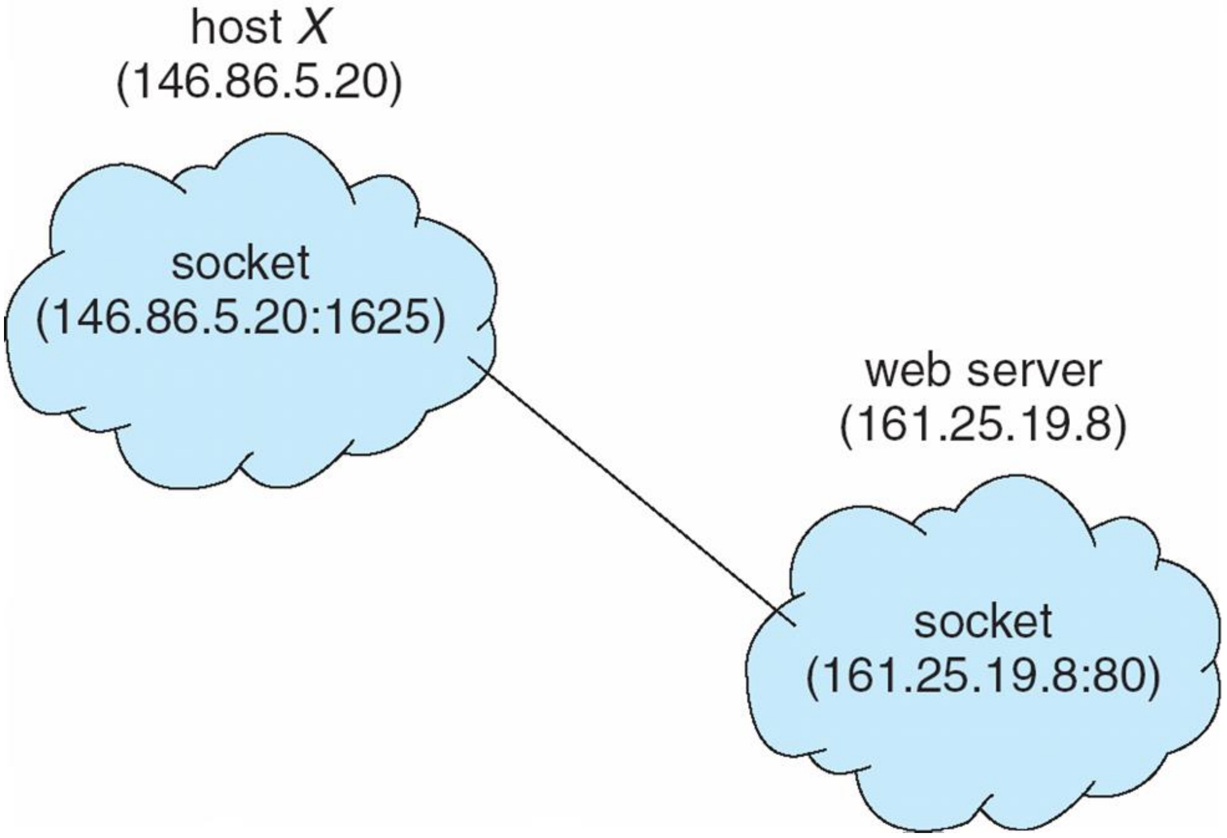


Computer Networks, Peterson and Davie

Sockets

- How can two processes on two different machines talk to each other on the web?
- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

Socket Communication



Connection Types

- Two types of connection (transport layer)
  - **Connection-oriented** (**TCP**)
  - **Connectionless** (**UDP**)

TCP Connections

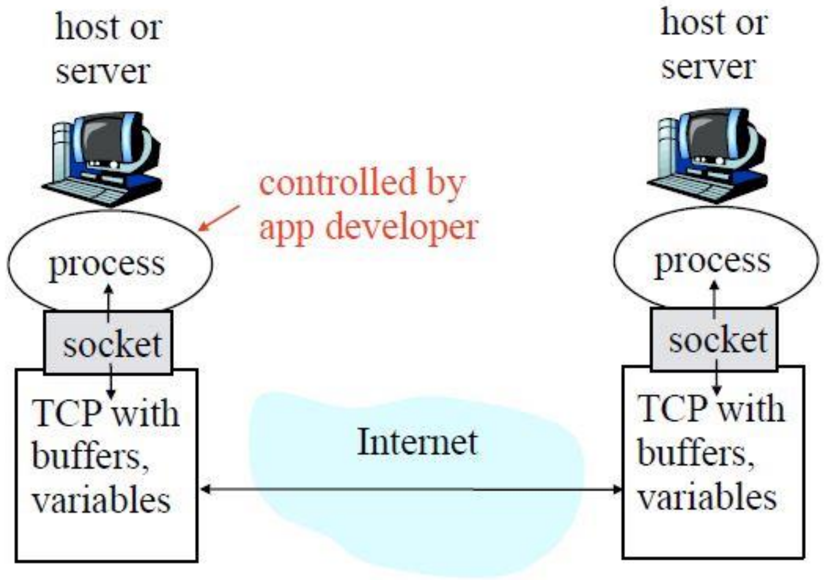
- Service
  - OSI Transport Layer
- Reliable byte stream (interpreted by application)
- 16-bit port space allows multiple connections on a single host
- Connection-oriented
  - – Set up connection before communicating
  - – Tear down connection when done

TCP Service

- Reliable Data Transfer
  - Guarantees delivery of all data
  - Exactly once if no catastrophic failures
- Sequenced Data Transfer
  - Guarantees in-order delivery of data
  - If A sends M1 followed by M2 to B, B never receives M2 before M1
- Regulated Data Flow
  - Monitors network and adjusts transmission appropriately
  - Prevents senders from wasting bandwidth
  - Reduces global congestion problems
- Data Transmission
  - Full-Duplex byte stream

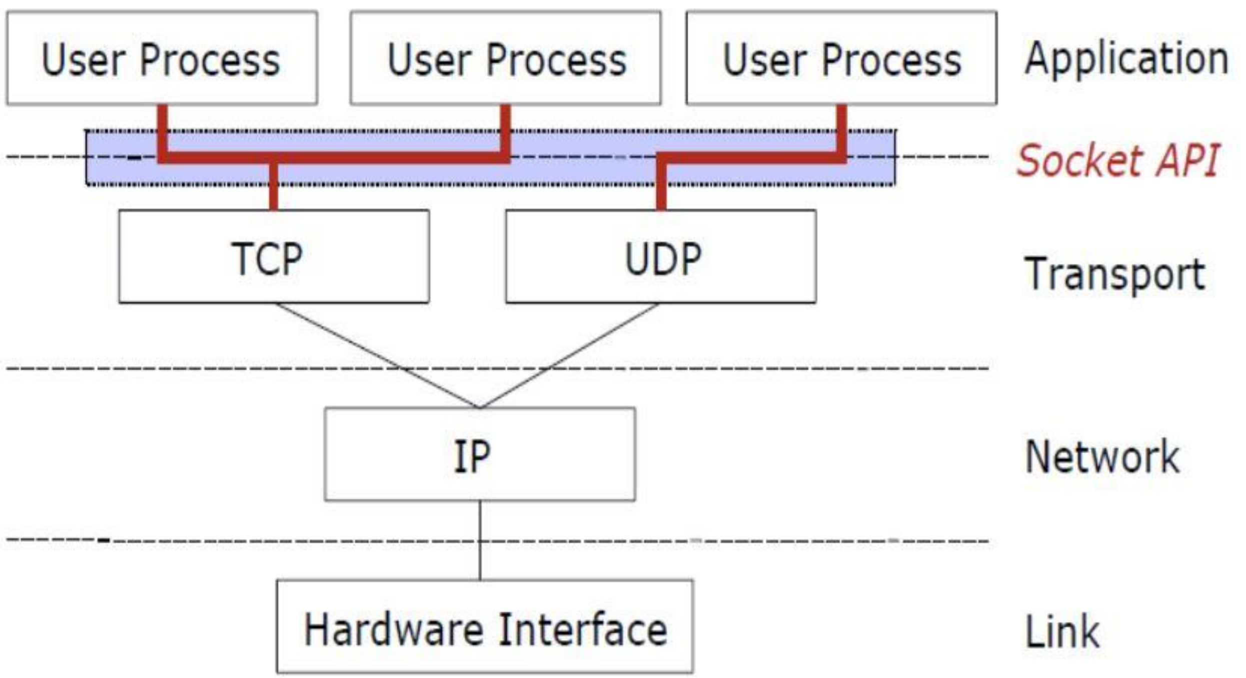
Sample TCP communication

- Transport Control Protocol (TCP)



www.cs.uluc.edu/class/fa07/cs438

TCP connection from OSI P.O.V.



www.cs.uluc.edu/class/fa07/cs438

TCP Connection Establishment

- Connection oriented (streams)
  - sd = socket(PF_INET, SOCK_STREAM, 0);

Default Protocol

```
#include <sys/socket.h>

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

Returns: file (socket) descriptor if OK, –1 on error

Domain	Description
AF_INET	IPv4 Internet domain
AF_INET6	IPv6 Internet domain (optional in POSIX.1)
AF_UNIX	UNIX domain
AF_UNSPEC	unspecified

Type	Description
SOCK_DGRAM	fixed-length, connectionless, unreliable messages
SOCK_RAW	datagram interface to IP (optional in POSIX.1)
SOCK_SEQPACKET	fixed-length, sequenced, reliable, connection-oriented messages
SOCK_STREAM	sequenced, reliable, bidirectional, connection-oriented byte streams

TCP Connection Establishment

- For the internet (PF_INET) this corresponds to TCP
- socket() returns a socket descriptor, an int similar to a file descriptor
- For a server, we need to associate a well-known address with the server's socket on which client requests will arrive
- Clients need a way to discover the address to use to contact a server
  - server reserves an address and register it in /etc/services
  - Register with a name service

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t len);
```


www.cs.uluc.edu/class/fa07/cs438

TCP Connection Establishment

- Use connect() on a socket that was previously created using socket():

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t len);
```

- If we're dealing with a connection-oriented network service, we need to create a connection between the socket of the process requesting the service (the client) and the process providing the service (the server)
- The address we specify with connect is the address of the server with which we wish to communicate. If sockfd is not bound to an address, connect will bind a default address for the caller.

TCP Connection Establishment

```
struct sockaddr_in {
  sa_family_t     sin_family;    /* address family */
  in_port_t       sin_port;      /* port number */
  struct in_addr  sin_addr;      /* IPv4 address */
  unsigned char   sin_zero[8];   /* filler */
};
```

www.cs.uluc.edu/class/fa07/cs438

TCP: Client

- socket() create the socket descriptor
- connect() connect to the remote server.
- read(),write() communicate with the server
- close() end communication by closing socket descriptor

TCP: Server

- socket() create the socket descriptor
- bind() associate the local address
- listen() wait for incoming connections from clients
- accept() accept incoming connection
- read(),write() communicate with client
- close() close the socket descriptor

Listen

- A server announces that it is willing to accept connect requests by calling the listen function
- The backlog argument provides a hint to the system regarding the number of outstanding connect requests that it should enqueue on behalf of the process

```
int listen(int sockfd, int backlog);
```

Accept Connections

- Once a server has called listen, the socket used can receive connect requests. We use the accept function to retrieve a connect request and convert it into a connection
- The file descriptor returned by accept is a socket descriptor that is connected to the client that called connect
- The original socket passed to accept is not associated with the connection, but instead remains available to receive additional connect requests

```
int accept(int sockfd, struct sockaddr *restrict addr,
           socklen_t *restrict len);
```

Returns: file (socket) descriptor if OK, –1 on error

Quiz 03!

FYR

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of fork() and exec()

- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- **`exec()`** usually works as normal – replace the running process including all threads

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
    1. Signal is generated by particular event
    2. Signal is delivered to a process
    3. Signal is handled by one of two signal handlers:
        1. default
        2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
    - **User-defined signal handler** can override default
    - For single-threaded, signal delivered to process

Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - I.e. **`pthread_testcancel()`**
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

Thread-Local Storage

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to `static` data
  - TLS is unique to each thread

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads