Assignment:

Read sections in Chapter 2 of Kurose-Ross.

We will cover in this order:
2.1, 2.7, 2.4, 2.3, 2.2, 2.6
Outline

[Introduce Application Layer]
[Socket Programming]
[Domain Name Service (DNS)]
[Standard Application-level Protocols]
  * email (SMTP)
  * HTTP
  * Video streaming

Some network applications

- e-mail
- web
- instant messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming stored video clips
- social networks
- voice over IP
- real-time video conferencing
- grid computing
- grid computing
Creating a network app

We write programs that
- run on (different) *hosts*
- communicate over network
- e.g., browser software communicates with web server software

We do not need to (and are not able to) write software for network-core devices
- Network-core devices do not run user code
- Limiting applications on end systems facilitates rapid app development, distribution

Client-server architecture

Most common paradigm

server:
- always-on host
- permanent IP address
- server farms for scaling

clients:
- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- Usually do not communicate directly with each other
Application-layer protocol defines:

- Types of messages exchanged, e.g., request, response
- Message syntax: what fields in messages & how fields are delineated
- Message semantics: meaning of information in fields
- Rules for when and how processes send & respond to messages

Public-domain protocols:
- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP, BitTorrent

Outline

- Introduce Application Layer
- Socket Programming
- Domain Name Service (DNS)
- Standard Application-level Protocols
  - email (SMTP)
  - HTTP
  - Video streaming
**Application Programming Interfaces**

- The TCP/IP protocol suite provides only the **protocols** that can be used by processes to communicate across a network.
- Though standarized, how these protocols are **implemented** is **system specific**
- Transport protocols and IP are typically implemented in the operating system.
- Hence, an API is needed to provide user processes with access to these protocols.

**“Low-level” Interfaces**

- Sockets - developed at Berkeley by Bill Joy and others (1981).
  - Incorporated into BSD Unix and all derivatives, also supported in Windows, Mac OS X, etc.
  - Key concept: make network access look much like file access.

- Technology-specific APIs
  - E.g., ATM LANs, special LANs for compute clusters, and so on.
“High-Level” APIs

- Remote Procedure Call (RPC)
  - “message passing” appears as “procedure calling”
  - client accesses remote server through procedure call interface.
  - typically implemented as a library atop sockets
- Object-oriented network programming
  - Java.net class library
  - OO middleware platforms such as CORBA
  - Java Remote Method Invocation (RMI)

Introduction to Sockets

- A socket is a kernel data structure, accessed via a file descriptor, that defines a bi-directional endpoint of communication
- Sockets are used as a basic building block for interprocess communication
- The kernel data structure
  - contains state information regarding the socket
  - has associated lists of send and receive buffers
  - accesses transport functions directly
socket() system call

- A socket is created using the socket() call, which takes as parameters:
  - protocol family
  - type of communication
  - specific protocol (often implicit)
- socket() returns a file descriptor
- A socket is created without a name
  - a socket “name” is a triple:
    - class (a.k.a. family), IP address, port number
  - a name can be affixed to the socket after it is created by invoking the bind() system call

Protocol Families

- Internet Domain (AF_INET)
  - interprocess communication among different hosts
  - socket name includes internet address and port
  - some port numbers are reserved for system use (see /etc/services)
- Unix Domain (AF_UNIX, aka AF_LOCAL)
  - UNIX system internal protocols
  - interprocess communication within same host
  - socket name is a file pathname
**INET Socket types**

- **datagram socket (SOCK_DGRAM)**
  - unreliable datagram communication
  - record boundary (fixed maximum) is preserved
  - implemented using the UDP protocol

- **stream socket (SOCK_STREAM)**
  - reliable virtual circuit communication
  - “record” boundary is not preserved
  - implemented using the TCP protocol

- **raw socket (SOCK_RAW)**
  - direct interface to the IP protocol
  - super-user (administrator) only, for new protocol development

**INET socket names**

- **Name Components**
  - Protocol family (AF_INET)
  - IP address
  - Port number (can be chosen by OS)

- **Socket must have a name in order to receive data**

- **Naming is done via bind() system call or implicit bind, discussed later.**
Socket programming basics

- Initial receiver (which we will call the server) must be *running* before the initial sender (the client) can (successfully) send anything to it.
- Server must have a *socket* through which it receives and sends segments
- Similarly, the client needs a socket
- The server socket is locally identified with (bound to) a *port number*
  - Analogous to the mailbox number
- Client *needs to know* the server’s IP address and port number.

Socket programming *with UDP*

UDP: no “connection” between client and server
- Sending application specifies the IP address and port of the destination
- The OS places these in the header of each outgoing datagram
- OS also places IP address and port of the sending socket in the header of each datagram
- Server can extract IP address, port of sender from received datagram

*Note:* the official terminology for a UDP packet is “datagram.” Some texts refer to “UDP segments,” but the term *segment* usually applies to parts of TCP data streams
A few notes about port numbers

- A port number is just a (16-bit) number; it has nothing to do with instances of physical hardware or even data structures.
- Port numbers 0 to 1023 are “well-known” and reserved for standardized applications; they are not available to regular users.
  - e.g., port 80?
- When invoking bind(), an application program can specify the port number to be bound to the socket. If already in use?
- A parameter of 0 means: let OS choose an available port number.

Socket examples

- Follows are a few simple examples of socket code.
- These examples are in C++ and can be copied, compiled and executed on CSE servers (arctic, black, ...)
- If you prefer Python, simple examples are provided in the Kurose-Ross text, and many examples/tutorials are available on the Internet.
Example: UDP receiver

// recv-dgram.cc -- datagram receive code
// After creating a port and binding a name to it, this program prints the port
// number, which is to be used by the sending side. In a loop, the program waits
// for a line of text then sends reply, rot13 encoded.

#include files omitted

#define BUFSIZE 356

int rot13 ( char *inbuf, char *outbuf )

void main ( )
{
    int sk ;  // socket descriptor
    sockaddr_in remote ;  // socket address for remote
    sockaddr_in local ;  // socket address for us
    char buf[BUFSIZE] ;  // buffer from remote
    char retbuf[BUFSIZE] ;  // buffer to remote
    int rlen = sizeof(remote) ;  // length of remote address
    int len = sizeof(local) ;  // length of local address
    int moredata = 1 ;  // keep processing or quit
    int msglen ;  // actual length of message

    CSE 422 - McKinley

    UDP receiver (cont.)

    // create the socket
    sk = socket(AF_INET,SOCK_DGRAM,0) ;

    // set up the socket
    local.sin_family = AF_INET ;  // internet family
    local.sin_addr.s_addr = INADDR_ANY ;  // wild card machine address
    local.sin_port = 0 ;  // let system choose the port

    // bind the name (address) to a port
    bind(sk,(struct sockaddr *)&local,sizeof(local)) ;

    // get the port name and print it out
    getsockname(sk,(struct sockaddr *)&local,&len) ;
    cout << "socket has port " << local.sin_port << "\n" ;
    // cout << "socket has addr " << local.sin_addr.s_addr << "\n" ;
UDP receiver (cont.)

while( moredata ) {
    // wait for a message and print it
    mesglen = recvfrom(sk,buf,BUFLEN,0,(struct sockaddr *)&remote, &rlen);
    buf[mesglen] = '\0';
    cout << buf << "\n";
    moredata = rot13(buf,retbuf);

    if( moredata ) {
        // send a reply, using the address given in remote
        sendto(sk,retbuf,strlen(retbuf),0,(struct sockaddr *)&remote, sizeof(remote));
    }
}

/* close the socket */
close(sk);

UDP receiver (cont.)

* Encode message using rot13 scheme.
 *
HANDS OFF
*

int rot13 ( char *inbuf, char *outbuf ) {
    int idx ;

    if( inbuf[0]=='.') return 0 ;

    idx=0 ;

    while( inbuf[idx]!='\0' ) {
        if( isalpha(inbuf[idx]) ) {
            if( (inbuf[idx]&31)<=13 )
                outbuf[idx] = inbuf[idx]+13 ;
            else
                outbuf[idx] = inbuf[idx]-13 ;
        } else
            outbuf[idx] = inbuf[idx] ;
        idx++ ;
    }
    outbuf[idx] = '\0';
    return 1 ;
}
Example UDP sender

// send-dgram.cc -- datagram sending code
// This program sends a message in a datagram, waits for, and prints a reply. The
destination machine name and destination port number are command line arguments.

#include files deleted
#define MSG1 "Have you heard about the new corduroy pillows?\nThey are making headlines!
#define MSG2 "."
#define BUFSIZE 356

void main ( int argc, char *argv[] )
{
    int sk ; // socket descriptor
    sockaddr_in remote ; // socket address for remote side
    char buf[BUFSIZE] ; // buffer for response from remote
    hostent *hp ; // address of remote host
    int msglen ; // actual length of the message

    // create the socket
    sk = socket(AF_INET,SOCK_DGRAM,0) ;
    // designate the addressing family
    remote.sin_family = AF_INET ;
    // get the address of the remote host and store
    hp = gethostbyname(argv[1]) ;
    memcpy(&remote.sin_addr,hp->h_addr,hp->h_length) ;
    remote.sin_port = atoi(argv[2]) ;

    // send the message to the other side
    sendto(sk,MSG1,strlen(MSG1),0,(struct sockaddr *)remote, sizeof(remote)) ;

    // wait for a response and print it
    msglen = read(sk,buf,BUFSIZE) ;
    buf[msglen] = '\0' ;
    cout << buf << "\n" ;

    // send message telling it to shut down
    sendto(sk,MSG2,strlen(MSG2),0,(struct sockaddr *)remote, sizeof(remote)) ;

    // close the socket and exit
    close(sk);
}
UDP observations & questions

- Both sender and receiver use datagram sockets
- Dest IP and port are explicitly included with the outgoing datagram.
- Question: how did the receiver (server) know the port number of the sender (client), if the client’s socket was never bound?
- Can multiple clients send to the same server program at the same port?

Socket programming with TCP

Client (active side) must
- connect to server (passive side)
- server process must first be already running
- server must have created a socket that receives the client’s connection request

Client contacts server by:
- creating a local TCP socket
- issuing a connect() system call, specifying
  - IP address of server socket
  - port number of server socket

Server actions:
- create and bind a name to the receiving socket
- Await connection request
- Upon receipt, TCP creates new socket for the server process to communicate with this client
  - allows server to talk with multiple clients on different sockets
  - all have same server port number
  - more on this later...
Stream socket – Passive side

- Await (listen for) connections
- Use listen() system call to prepare operating system for connection requests
- Parameters:
  - socket descriptor
  - backlog: defines the maximum length of the queue of pending connections (usually 5)

Passive side (cont.)

- Use the accept() system call
  - socket descriptor
  - name (family, host addr, port)
  - length of name
- Extracts the first connection on the queue of pending connections, creates a new socket, and allocates/returns a new file descriptor for the socket.
- Normally blocks, but this can be turned off
- Returns file descriptor of a new socket.
Stream socket – active side

- Initiate a connection on a socket
- Use the connect() system call
  - socket descriptor
  - name (family, host addr, port)
  - length of name
- Connect on a datagram socket (??)
- Errors
  - ETIMEDOUT
  - ECONNREFUSED
  - ENETDOWN or EHOSTDOWN
  - ENETUNREACH or EHOSTREACH

Transferring data

- Transmitting
  - write() as for files
  - send() uses flags to specify such requests as out-of-band transmission (MSG OOB)

- Receiving
  - read() as for files
  - recv() uses flags to specify such requests as examining data without reading it (MSG PEEK)
TCP Example (passive side)

// recv-stream.cc -- passive side stream socket example
void main ( )
{
    int    sk, sk2 ;              // socket descriptors
    sockaddr_in    remote ;      // socket address for remote
    sockaddr_in    local ;       // socket address for us
    char        buf[BUflen] ;    // buffer from remote
    char        retbuf[BUflen] ;  // buffer to remote
    int         rlen = sizeof(remote);  // length of remote address
    int         len = sizeof(local);   // length of local address
    int         moredata = 1 ;      // keep processing or quit
    int         mesglen ;          // actual length of message

    // create the socket
    sk = socket(AF_INET,SOCK_STREAM,0) ;
    // set up the socket
    local.sin_family = AF_INET ;    // internet family
    local.sin_addr.s_addr = INADDR_ANY ; // wild card machine address
    local.sin_port = 0 ;           // let system choose the port
    // bind the name (address) to a port
    bind(sk,(struct sockaddr *)&local,sizeof(local)) ;

    // get the port name and print it out
    getsockname(sk,(struct sockaddr *)&local,&len) ;
    cout << "socket has port " << local.sin_port << "\n" ;
    // tell OS to queue (up to 1) connection requests
    listen(sk, 1);

    // wait for connection request, then close old socket
    sk2 = accept(sk,(struct sockaddr *)0,(int *)0) ;
    close(sk);

    if(sk2 == -1)
        cout << "accept failed!\n" ;
    else { while( moredata ) {
        // wait for a message and print it
        mesglen = read(sk2,buf,BUflen);
        buf[mesglen] = '\0' ;
        cout << buf << "\n" ;
        moredata = rot13(buf,retbuf) ;
        if( moredata ) {
            // send a reply
            write(sk2,retbuf,strlen(retbuf));
        } }
    }
    close(sk2);
    exit(0);
}
TCP Example (active side)

// send-stream.cc -- active side stream socket example

#define MSG1 "Have you heard about the new corduroy pillows?\nThey are making headlines!"
#define MSG2 ""
#define BUFSIZE 356

void main ( int argc, char *argv[] )
{
    int sk; // socket descriptor
    sockaddr_in remote; // socket address for remote side
    char buf[BUFSIZE]; // buffer for response from remote
    hostent *hp; // address of remote host
    int msglen; // actual length of the message

    // create the socket
    sk = socket(AF_INET,SOCK_STREAM,0); // designate the addressing family
    remote.sin_family = AF_INET;
    // get the address of the remote host and store
    hp = gethostbyname(argv[1]);
    memcpy(&remote.sin_addr,hp->h_addr,hp->h_length);
    // get the port used on the remote side and store
    remote.sin_port = atoi(argv[2]);

    // connect to other side
    if(connect(sk, (struct sockaddr *)&remote, sizeof(remote)) < 0) {
        cout << "connection error!\n"
        close(sk);
        exit(1);
    }

    // send the message to the other side
    write(sk,MSG1,strlen(MSG1));
    // wait for a response and print it
    msglen = read(sk,buf,BUFSIZE);
    buf[msglen] = '\0';
    cout << buf << "\n";
    // send message telling it to shut down
    write(sk,MSG2,strlen(MSG2));
    // close the socket and exit
    close(sk);
}
Server example (passive side)

// recv-server.cc -- handles translation for multiple clients
void main ( )
{

  <variables as in earlier example>

  // create the socket
  sk = socket(AF_INET,SOCK_STREAM,0) ;
  // set up the socket
  local.sin_family = AF_INET ;       // internet family
  local.sin_addr.s_addr = INADDR_ANY ; // wild card machine address
  local.sin_port = 0 ;              // let system choose the port

  // bind the name (address) to a port
  bind(sk,(struct sockaddr *)&local,sizeof(local)) ;
  // get the port name and print it out
  getsockname(sk,(struct sockaddr *)&local,&len) ;
  cout << "socket has port " << local.sin_port << \\
       "\n";
  // tell OS to queue (up to 5) connection requests
  listen(sk, 5);

Server passive side (cont.)

// we loop forever, taking connections and forking new servers
for( ; ; ) {
  // wait for connection request
  sk2 = accept(sk, (struct sockaddr *)0, (int *)0) ;
  if(sk2 == -1) cout << "accept failed!\n" ;
  else {
    if( fork()==0 ) {
      // this is the child process ...
      close(sk) ;                      // sk is no longer needed
      while( moredata ) {
        // wait for a message and print it
        mesglen = read(sk2,buf,BUFLEN);
        buf[mesglen] = '\0' ;
        cout << buf << "\n" ;
        moredata = rot13(buf,retbuf) ;
        if( moredata ) { // send a reply
          write(sk2,retbuf,strlen(retbuf)) ;
        }
      }
      exit(0);
    }
    // this is the parent, so we no longer need sk2 ...
  }
}
}
Server example (active side)

// send-server.cc -- active side for server example

#define ENDMSG "\n"
#define BUFLEN 356

void main ( int argc, char *argv[] ) {
    int sk ;     // socket descriptor
    sockaddr_in remote ;   // socket address for remote side
    char buf1[BUFLEN] ;    // buffer for sending to remote
    char buf2[BUFLEN] ;    // buffer for response from remote
    hostent *hp ;          // address of remote host
    int msglen ;           // actual length of the message

    // create the socket
    sk = socket(AF_INET,SOCK_STREAM,0) ;
    // designate the addressing family
    remote.sin_family = AF_INET ;
    // get the address of the remote host and store
    hp = gethostbyname(argv[1]) ;
    memcpy(&remote.sin_addr,hp->h_addr,hp->h_length) ;
    // get the port used on the remote side and store
    remote.sin_port = atoi(argv[2]) ;

    // connect to other side
    if(connect(sk,(struct sockaddr *)&remote,sizeof(remote)) < 0) {
        cout << "connection error!\n" ;
        close(sk);
        exit(1);
    }
    // loop, reading input and sending to other side, until a single '.' is typed
    cin.getline(buf1,sizeof(buf1));
    while (buf1[0] != '.') {
        // send the message to the other side
        write(sk,buf1,strlen(buf1));
        // wait for a response and print it
        msglen = read(sk,buf2,BUFLEN) ;
        buf2[msglen] = '\0';
        cout << buf2 << "\n" ;
        // get next line of input
        cin.getline(buf1,sizeof(buf1));
    }
    // send (last) message telling it to shut down
    write(sk,buf1,strlen(buf1));
    // close the socket and exit
    close(sk);
}
Secure Socket Layer (SSL)

- More recently standardized as Transport Layer Security (TLS), but many still call it SSL
- SSL basically provides an encrypted TCP connection
- Implemented as application level software
- Cryptographic key handshake extends the TCP connection setup procedure
- Enables secure HTTP (HTTPS)
- Details discussed later in the course

TCP/socket summary and a question

- Active and passive sides use stream sockets
- Passive side does bind(), listen() and accept()
- Active side must connect() to passive side
- Once connected, either side can read and write.
- When the passive side accepts a connection, it creates a new socket, but with the same port number as the original socket.
- Question: So, if packets are arriving from different clients, but all with the same destination port number, how does the OS know which socket each belongs to?