EE/CS 145a

Lecture: Introduction to Network Programming

David Noblet
dnoblet@cs.caltech.edu

Overview

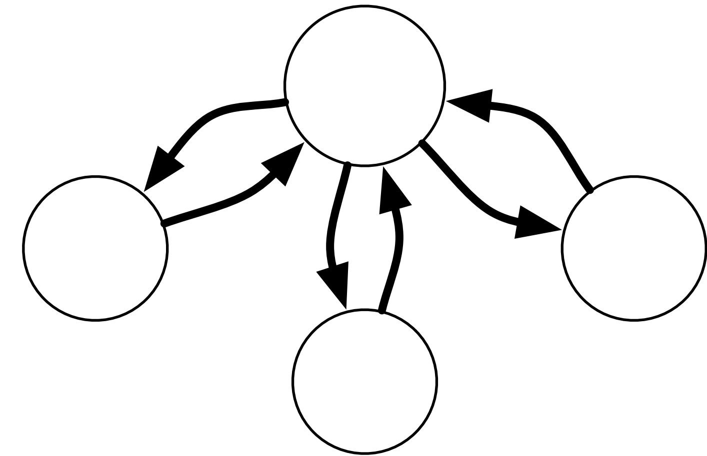
- There will be **four** lab assignments
- All the labs will involve programming; all the programming will be in the **C** programming language; each lab will make use of the **Sockets API**
- Although you may develop your code on any machine you choose, it **must compile and run** on the (Linux-based) CS machines
- This lecture is designed to provide you with an introduction to network programming and the environment (namely, Linux) in which you will be working for your labs
- The labs are intended to be more-or-less self-contained; so, be sure to ask questions if something is not clear

Motivation: Why Write Network Applications?

- To share information between remote locations
- To accumulate and share resources
  - i.e. Distributed storage/processing
- To provide redundancy and increase reliability
  - Isolation (it is easier for a server to fail than a CPU)
- To facilitate interoperability
  - Abstraction boundary between components

The Client-Server Model

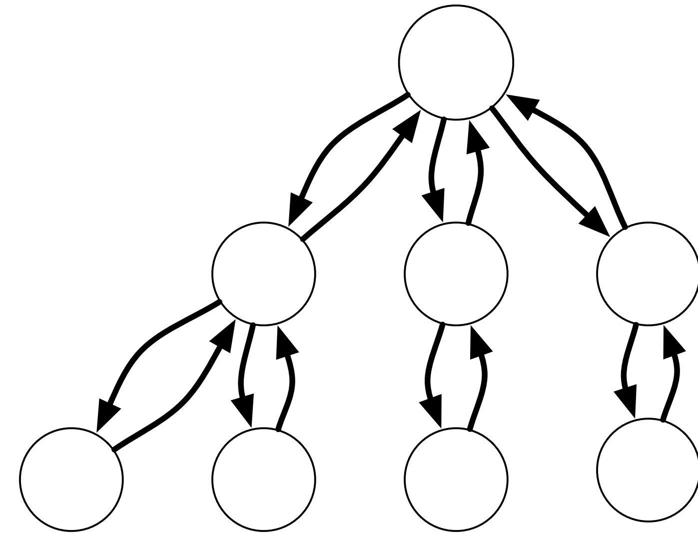
- The restaurant analogy
  - Servers advertise and provide service
  - Clients request and consume service

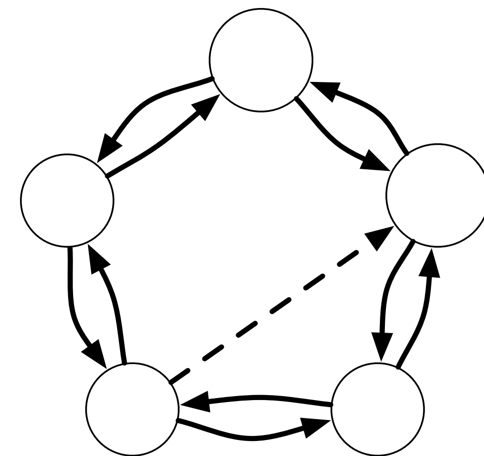


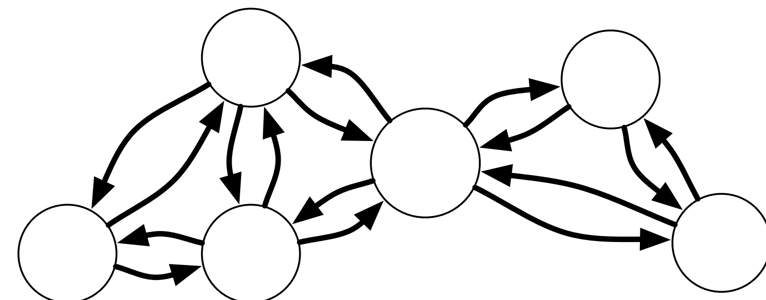
- Most network applications in wide use adhere to this model
  - Examples include: web browsing (HTTP), email (POP3/IMAP), file transfer (FTP), file sharing (CIFS)
- Design advantages
  - Simple, efficient (decisions can be made locally)

Alternative Models

- Decentralize and distribute
  - Address the reliability, scalability, and performance problems w/ client-server model
- Peer-to-peer (P2P)
  - Refers to a range of many, more specific, host role orientations
  - Hierarchical, DHT, Gossip
- Hybrid models



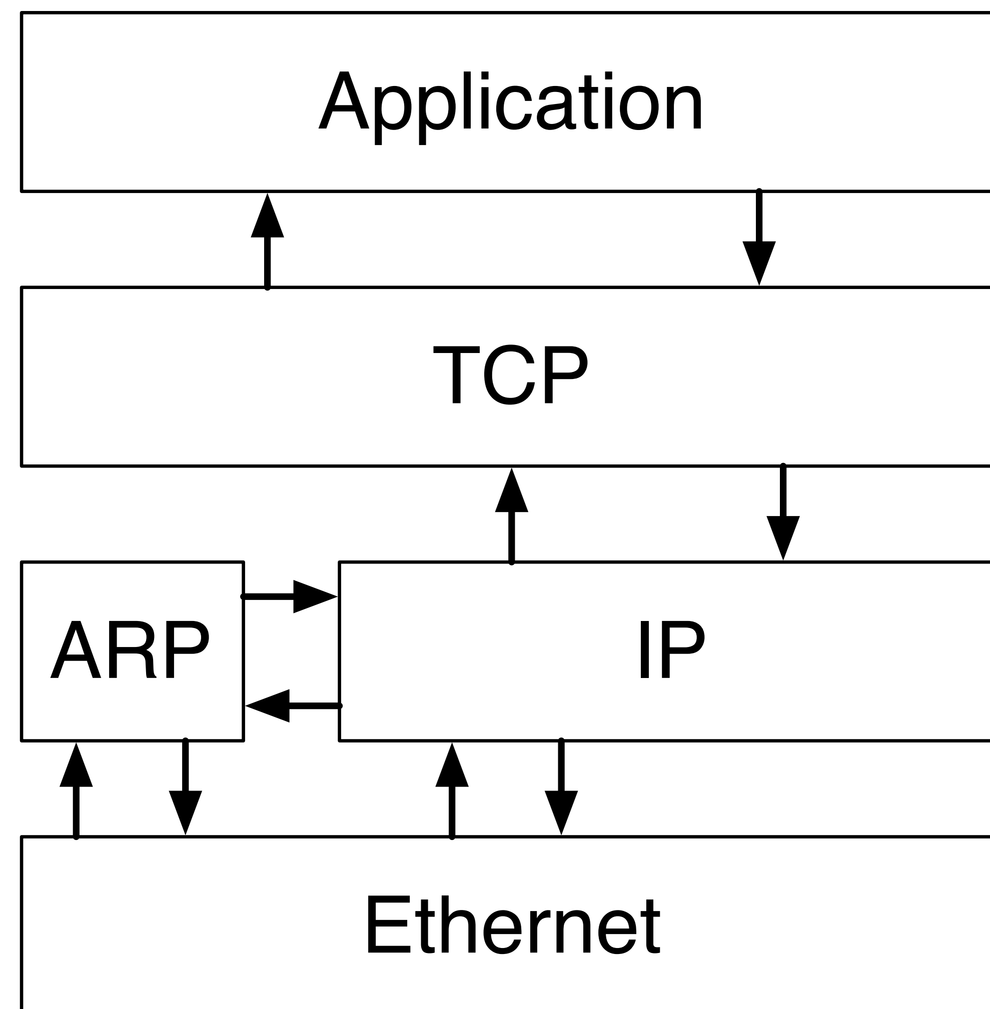




Network Protocol Stack

- Network software is often organized into “layers”
- Collectively, a particular combination of layers is known as a “stack”
- In the layered design pattern, each layer...
    - Advertises dependencies for the layer below
    - Exports an interface to the layer above
    - Provides a set of guarantees
- A layer may rely on the guarantees provided by the layer below to enforce its own guarantees

Example: A TCP/IP Stack (over Ethernet)



Transport Protocols

- The layer of the stack just above the network layer is often referred to as the *transport* layer
- Two of the most commonly used transport-layer protocols are:
  - **TCP** (Transmission Control Protocol)
  - **UDP** (User Datagram Protocol)
- Protocols at this layer provide guarantees that govern:
  - Connection orientation, multiplexing, message ordering, reliable delivery, and flow/congestion control

TCP & UDP: Protocol Guarantees

	UDP	**TCP**
Connection-orientation	Connectionless	Connection-oriented
Data Presentation	Datagram	Byte stream
Message Ordering	None	In-order
Reliable Delivery	No	Yes
Flow/Congestion Control	No	Yes

Probing the Network Environment

- Most modern operating systems have a network stack
    - Generally, this includes a TCP/IP implementation (although others may exist)
- Under Unix, we can easily probe aspects of our role in the network:
    - **ifconfig** -- Determine machine addresses (IP, ethernet, broadcast), net mask, MTU, link speed, among others
    - **ping** -- Diagnose network connectivity between hosts
    - **traceroute** -- Observe packet routing between hosts

Probing the Network Environment (ctd.)

- **netstat** -- Show connection states, routing tables, and routing statistics
- **arp** -- Display/modify IP-to-ethernet address translation table

Capturing Network Traffic

- Since Ethernet is a broadcast protocol, it is possible to “eavesdrop” on communication between other machines (provided you have sufficient access privileges on your machine)
- The following tools provide easy access to this functionality:
  - **tcpdump** -- Command-line dumping/filtering
  - **Wireshark** (formerly Ethereal) -- Graphical interface
- ... and justify the use of ssh for remote logins :-)

Demonstration: Command Use Examples

Sockets API (in C)

- Unix variants provide an interface to the transport layer of their protocol stack through the *sockets* API:
  - **socket** -- Allocate a new socket of a particular protocol type
  - **bind** -- Attach a socket to a particular address (i.e. an IP-port pair)
  - **listen**/**accept** -- Wait for incoming connections (if the protocol supports this)
  - **connect** -- Initiate a connection (if the protocol supports this)
  - **getsockopt/setsockopt** -- Get/set options available for the socket

Sockets API (in C) ctd.

- **send**/**sendto**/**sendmsg**/**write** -- Send data out on a socket
- **recv**/**recvfrom**/**recvmsg**/**read** -- Receive incoming data from a socket
- **shutdown**/**close** -- Release resources associated with the socket

- NOTE: Some of these functions (**read**, **write**, and **close**) are not specific to sockets and may be used with other file descriptors

Connection Example: Server

```
/* int main(int argc, char** argv) */
struct sockaddr_in sockaddr;
int err, opt, sock_listen, sock_conn;

// Check the user-supplied arguments
if (check_args(argc, argv))
{
    print_usage(stderr, APP_NAME);
    exit(1);
}

// Get the address & port from the user
fprintf(stderr,"Getting listen address...\n");
if((err = get_listen_address(argv, &sockaddr)) < 0)
{
    print_error(stderr, APP_NAME,
        strerror(-err));
    exit(1);
}

// Create a new TCP socket
fprintf(stderr,"Creating TCP socket...\n");
if((sock_listen = socket(AF_INET, SOCK_STREAM,
        PROTO_DEFAULT)) < 0)
{
    print_error(stderr, APP_NAME,
        strerror(errno));
    exit(1);
}
```

```
/* int get_listen_address(char** argv,
           struct sockaddr_in* sockaddr) */
int err = 0;
memset(sockaddr,0,sizeof(struct sockaddr_in));
sockaddr->sin_family = AF_INET;
sockaddr->sin_port =
    htons(strtoul(argv[ARG_PORT], NULL, 10));
if(errno)
{
    return -errno;
}
if((err = inet_pton(AF_INET, argv[ARG_HOST],
        &sockaddr->sin_addr)) < 0)
{
    return -err;
}

return err;
```

Connection Example: Server (ctd.)

```
/* int main(int argc, char** argv) continued...*/
// Try to allow immediate reuse of the address
opt = 1;
setsockopt(sock_listen, SOL_SOCKET, SO_REUSEADDR,
    &opt, sizeof(opt));

// Bind the socket to the supplied address and
//   port
fprintf(stderr,"Binding to address...\n");
if(bind(sock_listen, (struct sockaddr*)
        &sockaddr, sizeof(sockaddr)) < 0)
{
    print_error(stderr, APP_NAME,
        strerror(errno));
    exit(1);
}

// Listen for incoming connections
fprintf(stderr, "Set the socket to listen for "
    "incoming connections...\n");
if(listen(sock_listen,LISTEN_QUEUE_SIZE) < 0)
{
    print_error(stderr, APP_NAME,
        strerror(errno));
    exit(1);
}

// Accept a pending connection request
fprintf(stderr,"Waiting to accept pending "
```

```
/* int main(int argc, char** argv) continued...*/
    "request...\n");
if((sock_conn = accept(sock_listen, NULL, NULL))
        < 0)
{
    print_error(stderr, APP_NAME,
        strerror(errno));
    exit(1);
}

// Read some data
fprintf(stderr,"Waiting for data to read...\n");
if((err = read_question(sock_conn, NULL, 0)) < 0)
{
    print_error(stderr, APP_NAME,
        strerror(-err));
    exit(1);
}

// Send a response
fprintf(stderr,"Sending response...\n");
if((err = write_answer(sock_conn, ANSWER_STRING))
        < 0)
{
    print_error(stderr, APP_NAME,
        strerror(-err));
    exit(1);
}
```

Connection Example: Server (ctd.)

```
/* int read_question(int fd, char* qbuf,
       int qlen) */
int err = 0, idx = 0;
char cbuf;

while((err = read(fd,&cbuf,1)) == 1)
{
    if(qbuf)
    {
        qbuf[idx++] = cbuf;
        idx %= qlen;
    }

    if(cbuf == '?')
    {
        if (qbuf)
            qbuf[idx] = 0;
        return 0;
    }
}

return (errno == 0) ? -EINVAL : -errno;
```

```
/* int write_answer(int fd, const char*
       answer) */
int err = 0;
int len;

len = strlen(answer);

if((err = write(fd, answer, len)) != len)
    return -errno;

return 0;
```

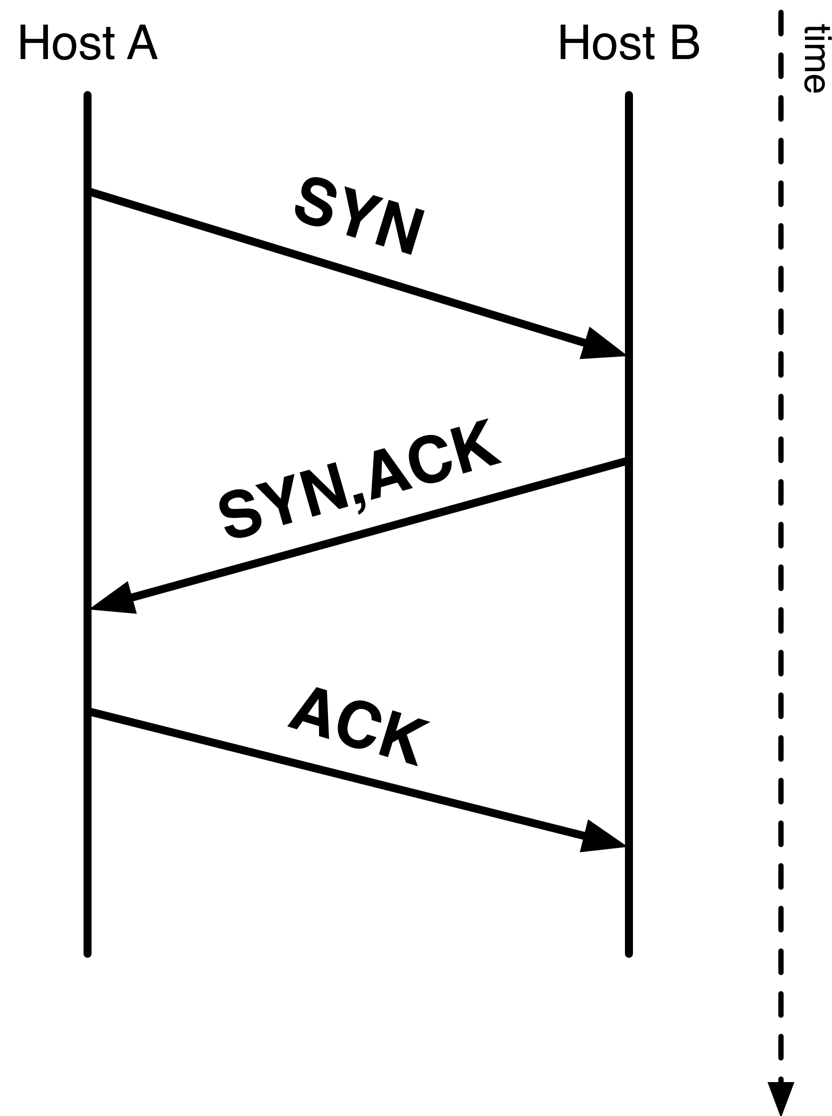
Connection Example: Client

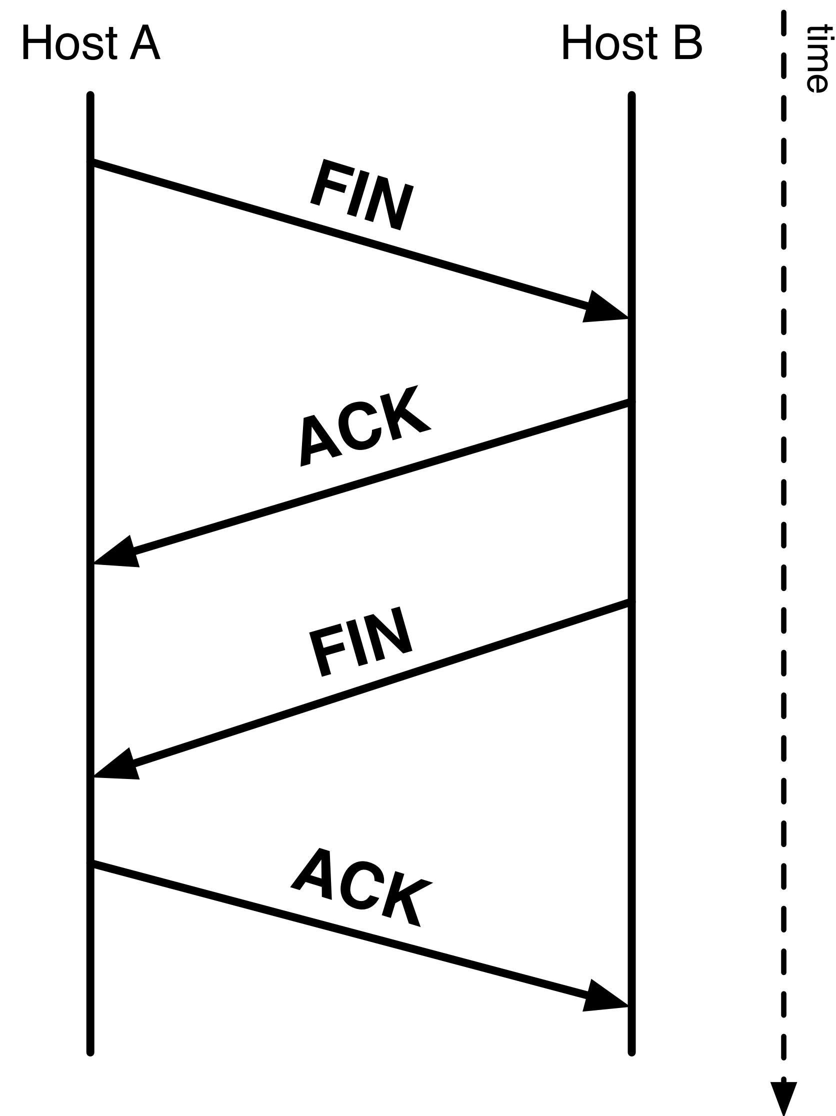
```c
/* int main(int argc, char** argv)*/
struct sockaddr_in sockaddr;
int err, opt, sock_conn;
char responsebuf[LARGE_BUFFER_SIZE];

// Check the user-supplied arguments
...

// Get the server address & port from the user
...

// Create a new TCP socket
fprintf(stderr,"Creating TCP socket...\n");
if((sock_conn = socket(AF_INET, SOCK_STREAM,
        PROTO_DEFAULT)) < 0)
{
    print_error(stderr, APP_NAME,
        strerror(errno));
    exit(1);
}

// Try to allow immediate reuse of the address
opt = 1;
setsockopt(sock_conn, SOL_SOCKET, SO_REUSEADDR,
    &opt, sizeof(opt));

// Connect to the server
fprintf(stderr,"Connecting to the server...\n");
if(connect(sock_conn, (struct sockaddr*)
```

```c
/* int main(int argc, char** argv) continued...*/
        &sockaddr, sizeof(sockaddr)) < 0)
{
    print_error(stderr, APP_NAME,
        strerror(errno));
    exit(1);
}

// Send a response
fprintf(stderr,"Writing some data...\n");
if((err = write_question(sock_conn,
        QUESTION_STRING)) < 0)
{
    print_error(stderr, APP_NAME,
        strerror(-err));
    exit(1);
}

// Read some data
fprintf(stderr,"Reading the response...\n");
if((err = read_answer(sock_conn, responsebuf,
        sizeof(responsebuf))) < 0)
{
    print_error(stderr, APP_NAME,
        strerror(-err));
    exit(1);
}
```

Demonstration: Connection Example

Anatomy of Making/Breaking a TCP Connection





Summary of Other Useful Functions (in C)

- **gethostbyname** -- Resolve a host name to an IP address
- **gettimeofday** -- Get the current system time
- **select** -- Wait on a set of file descriptors for an event (or timeout)
- **sigaction/sigprocmask** -- Set a signal handler or manipulate the signal mask (esp. useful for handling SIGPIPE signals)
- **assert** -- Assert that an expression evaluates to TRUE (in the C sense)

Network Programming Tips

- Use ‘assert’ when you make an assumption -- this avoids many errors in the first place
- Add plenty of logging (I suggest sending output to stderr); write macros that allow you to enable/disable it using “#define” compiler directives
- For hard-to-diagnose problems, use network monitoring software (tcpdump, Wireshark, etc.), if possible, to get a better idea of what packets are actually being sent
- Do not rely on a specific ordering or timing of events between hosts just because it is probable; be prepared to handle the worst case scenario
- Causality is your friend (i.e. a message will never be received before it is sent)

Common Pitfalls

- The firewall/NAT is getting in the way
- Listening on the wrong network interface
- Using wrong port ranges (0-1023 typically require administrative privileges)
- Forgetting to switch to and from network byte ordering (i.e. when specifying addresses, ports, etc)
- Reading/writing too little or too much (esp. w/ TCP)
- Signals interrupting system calls

Common Pitfalls (ctd.)

- Forgetting/neglecting to check for errors on function return
- Comparing local and remote time-stamp values

The Two Generals Problem

- There are two armies, each with a general, preparing to attack a city
- For the attack to succeed, they must coordinate on a plan
- Due to their respective positions, they may only communicate via messenger
- However, a messenger from one army must travel some distance through enemy territory to reach the other army (so he might be captured or killed)
- And there is no way to communicate with a messenger en-route
- Is there a protocol that will guarantee that both generals will attack?

Administrative Details

- To submit labs, you will need a CS account; if you do not have one, you may request one by filling out the form here:
  - http://www.cs.caltech.edu/cgi-bin/sysadmin/account_request.cgi
- There are no designated “lab hours” -- you must complete the labs on your own time (though you will have access to the CS Instructional Lab, Jorgensen 154)
- The times for my office hours have not been fixed yet; I’m open to suggestions as to when might be best for everyone

Style Notes for Lab Assignments

- Comment all major functions, data structures, and complex conditionals
- Provide robust error-handling code; error cases must be handled for all functions that can have an error result
- Compile with all warnings enabled ("-Wall" if you are using gcc) and eliminate them before submission
- I suggest using a tool like "make" (or, better yet, OMake -- http://omake.metaprl.org) to help automate the compilation and testing of your lab projects
- **ALWAYS** provide a "readme" file, explaining how to compile and run your code