Outline

• APIs – Motivation
• Sockets
• C Socket APIs
• Tips for programming
What is an API?

• **API** – stands for Application Programming Interface
What is an API?

- **API** – stands for **Application Programming Interface**.
- **Interface to what?** – *In our case, it is an interface to use the network.*
What is an API?

• **API** – stands for Application Programming Interface.

• **Interface to what?** – In our case, it is an interface to use the network.

• **A connection to the transport layer.**
What is an API?

- **API** – stands for Application Programming Interface.
- **Interface to what?** – In our case, it is an interface to use the network.
- **A connection to the transport layer.**

- **WHY DO WE NEED IT?**
Need for API

• One Word - Layering
• Functions at transport layer and below very complex.
• E.g. Imagine having to worry about errors on the wireless link and signals to be sent on the radio.
• Helps in code reuse.
Layering Diagrammatically

Application

API

System Calls

LAN Card

Radio
What is a socket then?

- What is a socket?
Introduction

- What is a socket?
- It is an abstraction that is provided to an application programmer to send or receive data to another process.
Introduction

• What is a socket?
• It is an abstraction that is provided to an application programmer to send or receive data to another process.
• Data can be sent to or received from another process running on the same machine or a different machine.
• In short, it is an end point of a data connection.
Socket – An Abstraction

Adapted from http://www.troubleshooters.com/codecom/sockets/
Sockets

- It is like an endpoint of a connection
- Exists on either side of connection
- Identified by IP Address and Port number
- E.g. Berkeley Sockets in C
  - Released in 1983
  - Similar implementations in other languages
Engineers working on Sockets!!!

http://www.fotosearch.com/MDG238/frd1404/
Ports

• Sending process must identify the receiver
  – Address of the receiving end host
  – Plus identifier (port) that specifies the receiving process

• Receiving host
  • Destination address uniquely identifies the host

• Receiving process
  • Host may be running many different processes

• Destination port uniquely identifies the socket
  • Port number is a 16-bit quantity
Port Usage

• Popular applications have “well-known ports”
  • E.g., port 80 for Web and port 25 for e-mail
  • Well-known ports listed at http://www.iana.org

• Well-known vs. ephemeral ports
  • Server has a well-known port (e.g., port 80)
  • By convention, between 0 and 1023; privileged
    • Client gets an unused “ephemeral” (i.e., temporary) port
    • By convention, between 1024 and 65535

• Flow identification
  • The two IP addresses plus the two port numbers
    • Sometimes called the “four-tuple”
  • Underlying transport protocol (e.g., TCP or UDP)
  • The “five-tuple”
Ports (Main Points)

- Not related to the physical architecture of the computer.
- Just a number maintained by the operating system to identify the end point of a connection.
TCP (stream) sockets

- Also known as SOCK_STREAM
- TCP is a connection-oriented byte-stream protocol
  - During data packet transmission, no packetization and addressing required by application.
  - Formatting has to be provided by application.
  - Two or more successive data sends on the pipe connected to socket may be combined together by TCP in a single packet.
  - E.g. Send “Hi” then send “Hello Nikhil” is combined by TCP to send as “HiHello Nikhil”
**UDP (datagram) sockets**

- Also known as **SOCK_DGRAM**
- **UDP is connectionless and packet-oriented.**
  - Info sent in packet format as needed by app.
  - Every packet requires address information.
  - Lightweight, no connection required.
  - Overhead of adding destination address with each packet at the application layer. (Can be eliminated by “connecting” – see later)

- Distinction in the way these sockets are used by different hosts – client and server.
Client – Server Architecture

Flow in client-server model

- http://www.process.com/tcpip/tcpware57docs/Programmer/fig1-2.gif
Typical Client Program

• Prepare to communicate.
  • Create a socket.
  • Determine server address and port number.
  • Initiate the connection to the server (TCP).

• Exchange data with the server.
  • Write data to the socket.
  • Read data from the socket.

• Note, single socket supports both reading and writing.
  • Manipulate the data (e.g., display email, play music)

• Close the socket.
Typical Server Program

- Prepare to communicate
  - Create a socket
  - Associate local address and port with the socket
- Wait to hear from a client (passive open)
  - Indicate how many clients-in-waiting to permit
  - Accept an incoming connection from a client
- Exchange data with the client over new socket
  - Receive data from the socket
  - Do stuff to handle the request (e.g., get a file)
  - Send data to the socket
  - Close the socket
- Repeat with the next connection request
One Server One port Many clients

- Consider a webserver running on port 80.
- All clients connect to the same port number.
- How do you distinguish between clients?
One Server One port Many clients

- Consider a webserver running on port 80.
- All clients connect to the same port number.
- How do you distinguish between clients?
- Source IP Address!

- How do you distinguish between multiple connections from the same IP Address?
One Server One port Many clients

• Consider a webserver running on port 80.
• All clients connect to the same port number.
• How do you distinguish between clients?
• Source IP Address!

• How do you distinguish between multiple connections from the same IP Address?
• OS uses the incoming packet’s source IP address and port number to distinguish.
Going into the APIs

- Will look into programming from now on.
- Stop me when not clear.
- Or if I am too fast.
- Or if you have never seen something and I am assuming you have!
- Most examples from “Beej’s guide” – link posted online.
- More examples in there. You should look into them.
  - Helpful for the project.
Creating a socket

- Operation to create a socket
  - `int socket(int domain, int type, int protocol)`
  - Returns a descriptor (or handle) for the socket
  - Originally designed to support any protocol suite
- Domain: protocol family
  - Use PF_INET for the Internet
- Type: semantics of the communication
  - SOCK_STREAM: reliable byte stream
  - SOCK_DGRAM: message-oriented service
- Protocol: specific protocol
  - UNSPEC: unspecified. No need for us to specify, since PF_INET plus SOCK_STREAM already implies TCP, or SOCK_DGRAM implies UDP.
- Used by both server and client to create socket.
Connecting to server

- Establishing the connection
  - `int connect(int sockfd, struct sockaddr *server_address, socketlen_t addrlen)`
  - Arguments: socket descriptor, server address, and address size
  - Returns 0 on success, and -1 if an error occurs
- `sockfd` stands for socket file descriptor.
  - Remember everything in Unix is a file.
- What is `sockaddr`?
  - `struct` to store the IP address and port number you want to connect to.
**Struct sockaddr_in**

- Struct sockaddr_in has information about the destination IP address and port.
  - Same size as sockaddr.
- Must be used in the following way.
- Use AF_INET in sockaddr and not PF_INET.

```c
int sockfd;
struct sockaddr_in dest_addr; // will hold the destination addr
sockfd = socket(PF_INET, SOCK_STREAM, 0); // do some error checking!
dest_addr.sin_family = AF_INET; // host byte order
dest_addr.sin_port = htons(Destination_PORT); // short, network byte order
dest_addr.sin_addr.s_addr = inet_addr(Destination_IP);
memset(&(dest_addr.sin_zero), '\0', 8); // zero the rest of the struct
// don't forget to error check the connect()!
connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
```
Byte Ordering

• The networking API provides us the following functions:
  • uint16_t htons(uint16_t host16bitvalue);
  • uint32_t htonl(uint32_t host32bitvalue);
  • uint16_t ntohs(uint16_t net16bitvalue);
  • uint32_t ntohl(uint32_t net32bitvalue);

• Use for all 16-bit and 32-bit binary numbers (short, int) to be sent across network
• ‘h’ stands for “host order”
• These routines do nothing on big-endian hosts
IP Addresses

- IP Addresses should be in network format in a packet.
- We need to convert between ascii (dot format) and network format.
- Accomplished by inet_aton and inet_ntoa

```c
struct sockaddr_in antelope;
char *some_addr;
inet_aton("10.0.0.1", &antelope.sin_addr); // store IP in antelope
some_addr = inet_ntoa(antelope.sin_addr); // return the IP
printf("%s\n", some_addr); // prints "10.0.0.1"
```
Sending Data

- Sending data
  - `ssize_t write(int sockfd, void *buf, size_t len)`
  - Arguments: socket descriptor, pointer to buffer of data to send, and length of the buffer
  - Returns the number of characters written, and -1 on error

- Receiving data
  - `ssize_t read(int sockfd, void *buf, size_t len)`
  - Arguments: socket descriptor, pointer to buffer to place the data, size of the buffer
  - Returns the number of characters read (where 0 implies “end of file”), and -1 on error

- Closing the socket
  - `int close(int sockfd)`
Sending and Receiving (contd)

- Note: instead of using `write()`, you can instead use `send()`, which is intended for use with sockets.
  - Only difference is `send()` takes one additional argument of flags, which for most purposes don’t matter
- Similarly, instead of using `read()`, you can instead use `recv()`.
  - Again, only difference is one additional argument of flags
- Important to realize they’re basically equivalent, since you see both pairs of calls used (sometimes intermingled).
Example

char *msg = "I was here!";
int len, bytes_sent; ...
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);

• If the return value is -1 there is some error.
• If return value is less than the length of the message, it means the whole
  message was not sent for some reason.
• Then resend the remaining message.

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;
}
Server – Passive listening

- Passive open
  - Prepare to accept connections
  - … but don’t actually establish one
  - … until hearing from a client
- Hearing from multiple clients
  - Allow a backlog of waiting clients
  - ... in case several try to start a connection at once
- Create a socket for each client
  - Upon accepting a new client
  - ... create a new socket for the communication
Preparing a socket

- Bind socket to the local address and port number
  - `int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen)`
  - Arguments: socket descriptor, server address, address length
  - Returns 0 on success, and -1 if an error occurs
- Define how many connections can be pending
  - `int listen(int sockfd, int backlog)`
  - Arguments: socket descriptor and acceptable backlog
  - Returns 0 on success, and -1 on error
Accepting a connection

• Accept a new connection from a client
  • `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)`
  • Arguments: socket descriptor, structure that will provide client address and port, and length of the structure.
• Returns descriptor for a new socket for this connection.
• Accept will block the process if there are no clients trying to connect.
Example

```c
int sockfd, new_fd;
struct sockaddr_in my_addr; // my address information
struct sockaddr_in their_addr; // connector’s address information
int sin_size; // size of sockaddr
sockfd = socket(PF_INET, SOCK_STREAM, 0); my_addr.sin_family = AF_INET; // host byte order
my_addr.sin_port = htons(MYPORT); // short, network byte order
my_addr.sin_addr.s_addr = INADDR_ANY; // auto-fill with my IP
memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct
bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
listen(sockfd, BACKLOG);
sin_size = sizeof(struct sockaddr_in);
new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
```
Datagram sockets

- Datagram sockets may be used with/without `connect`.
  - Connecting a data socket does not create a connection.
    - Only fills in the address everytime you use a `send()`.
  - In this case, use `sendto()` and `recvfrom()`.
Advanced Aspects

• A general program may have many sockets open.
• Also it could have other sources of input like stdin or timers.
• What options does a program have for keeping a check on all these sources?
  • Polling
    • Very inefficient – Don’t use.
  • Using **select()**
    • Efficient and preferred method.
Select()

• Select()
  • Wait on multiple file descriptors/sockets and timeout
  • Application does not consume CPU while waiting
  • Return when file descriptors/sockets are ready to be read or written or they have an error, or timeout exceeded

• Disadvantages
  • Does not scale to large number of descriptors/sockets
  • More awkward to use than it needs to be
Select() - contd

FD_ZERO(fd_set *set) – clears a file descriptor set
FD_SET(int fd, fd_set *set) – adds fd to the set
FD_CLR(int fd, fd_set *set) – removes fd from the set
FD_ISSET(int fd, fd_set *set) – tests to see if fd is in the set

- int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
- The macros are used to set, clear and check conditions on the fds in the set.
`#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#define STDIN 0 // file descriptor for standard input`

```c
int main(void) {
    struct timeval tv;
    fd_set readfds;
    tv.tv_sec = 2;
    tv.tv_usec = 500000;
    FD_ZERO(&readfds);
    FD_SET(STDIN, &readfds);
    // don’t care about writefds and exceptfds:
    select(STDIN+1, &readfds, NULL, NULL, &tv);
    if (FD_ISSET(STDIN, &readfds))
        printf("A key was pressed!\n");
    else
        printf("Timed out.\n");
    return 0;
}
```
Some Programming Hints

• Check Beej’s guide (it is on the syllabus page)
  • Has information on all the APIs available.
  • Also tells you which header files to include for the different APIs.
• Also, it is best to catch errors using the returning values of the APIs.
  • Makes things easier to debug
  • And you know where the program fails.

```c
if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1) {
    perror("bind");
    exit(1);
}
```
Perror and strerror

- **Use perror and strerror**
- **If there is an error errno variable is set to a value and that gives more info on the error.**
- **Ofcourse there are also the man pages!**

```c
int s;
s = socket(PF_INET, SOCK_STREAM, 0);
if (s == -1) { // some error has occurred
  // prints "socket error: " + the error message:
  perror("socket error");
}
// similarly:
if (listen(s, 10) == -1) {
  // this prints "an error: " + the error message from errno:
  printf("an error: %s\n", strerror(errno));
}
```
Network Programming Tips (contd)

• **How to check if particular port is listening**
  • Windows – use netstat
    • netstat -an
  • Linux – use nmap
    • nmap -sT -O localhost

• **Tip: Use port numbers greater than 1024.**
• Server can’t bind because old connection hasn’t yet gone away.
  • Use setsockopt with the SO_REUSEADDR option.
• Not knowing what exactly gets transmitted on the wire
  • Use **tcpdump** or **Ethereal** ([www.ethereal.com](http://www.ethereal.com))
• **Check RFCs if in doubt about protocols.**
  • **http://www.ietf.org/rfc**