Goal: Networked PONG

Debouncing methods

1) Periodically check inputs every X milliseconds
   Note when input has changed
   If changed input holds its value through N checks, consider it changed
   See http://www.dattalo.com/technical/software/pic/debounce.html

2) Periodically check inputs
   If input has changed, delay for X milliseconds waiting for bounces to settle

3) Periodically check inputs
   If change of input, note time of change
   If no change for X milliseconds, consider it changed
   See http://www.arduino.cc/en/Tutorial/Debounce

IP networking 101

MAC (Media Access Control) addresses
- 48 bits
  - 24 bits of OUI (identifies manufacturer)
  - 24 bits of NIC (unique for each device of manufacturer)
- Expressed at six pairs of hexadecimal digits
  - Pairs correspond to octets
  - Pairs separated by colons or dashes
- Last bit of first octet is 0 for unicast addresses
  - Least significant bit or most significant byte
  - However, in Ethernet least significant bits goes first

Uses in CSCI 373
- 48 bit – Ethernet, 802.11 wireless, bluetooth
- 64 bit – ZigBee
- Sometimes, you'll have to set a MAC address

Normal uses
- Devices are manufactured with fixed address
- Look at back of computer
- `/sbin/ifconfig`
- `/sbin/arp`

MAC exercise
- Use `ifconfig` and `arp` to check out Ethernet addresses within the room

Switches
- Make a complex network look like a simple one

IP addresses

- IPv4 – 32 bits
- IPv6 – 128 bits

Division of bits

- Classful addressing -- A, B, and C
- CIDR

Routing

- Routing tables
  - Today complex routing tables are stored in high-priced routers
  - Routing information exchanged via SNMP
- "Private" networks
  - Do not connect to internet
  - Safest for use in embedded systems
  - 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16
  - Link-local address – 169.254.0.0/16

Useful programs for looking at routes

- `/sbin/ifconfig`
- `/sbin/route`

ARP (Address Resolution Protocol)

- Associates local IP addresses with MAC addresses
  - without the use of a table
- Some embedded systems do not support ARP
  - this is a problem with "clients" wishing to connect to the embedded system
- Variations of / problems with ARP
  - Proxy ARP
  - ARP spoofing

DHCP (Dynamic Host Configuration Protocol)

- Assigns IP addresses to local network computers
- May not be supported for embedded systems
- Computer Science DHCP server runs on `penrose.cs.unca.edu`
  - but not used for any Linux systems

DNS (Domain Name System)

- Distributed and redundant database
- Types of servers
  - Root servers
  - Authoritative servers (master and slave)
  - Caching servers
- Resource records
- Computer Science DNS server files
  - cs.unca.edu
  - csci.cs.unca.edu
  - 152.18.69.0/24
- Looking from the command line
  - `/usr/bin/dig`
  - `/usr/bin/nslookup`
  - `/usr/bin/host`

Exercise

- Look up root servers
- Loop up all records for unca.edu and cs.unca.edu

Port numbers
- Applications are identified by 16-bit port numbers
  - By ancient convention, numbers less than 1024 are “privileged”

TCP (Transmission Control Protocol)
- For “reliable” communication with sessions
- Data is acknowledged and retransmitted
- Programmer interfaces
  - Berkeley socket interface
    - Servers – `socket`, `bind`, `listen`, `accept`
    - Clients – `socket`, `connect`
    - Server & client – `read`/`write`, `close`
    - host/port lookup – `gethostbyname`, `getservyname`
  - Python sockets
  - java.net socket interface
- Peculiarities and problems
  - Network and host order – `htons`, `htonl`, `ntohs`, `ntohl`
    - Network byte order is big-endian
    - Intel is little-endian
  - It's just a sequence of bytes
    - Often “lines” are ended with CR+LR
      - and `"\x0D\0x0A"` may not be `"\n\r"`
    - Do not expect that one `write` equals one `read`
  - Maintaining several active sessions
    - For a server, for every client there is one socket
  - Looking like a terminal
    - Pseudo-terminal
  - Security

Exercise
- Using example programs from Wikipedia (in C or Java), create a client and server
  - Have the client and server program start with an interesting greeting
- Connect to servers on other computers written by other students
  - You can test with `telnet` *`machine portnumber`*

Getting Starter

1) Make sure the firewall has been opened.
2) Get your own port number.
3) Learn how to use `nc` to check if ports are open.
   To start a server
   `nc -l portnumber`
   To start a client
   `nc hostname portnumber`

First attempt – One connection server

Download the first server starter program, start it, and test with `nc`. Also, test connections to the servers of your fellow classmates.

Presently, the program just repeats numbers typed from a single remote connection. Modify the server so that if produces more interesting output. Interesting output exhibits hysteresis, that is, the output depends not only on the present input number, but also on past input numbers. A simple example of this would be writing both the present and last input numbers back to the client.

Second attempt – One connection of one line at a time

Modify your program so that it accepts several connections serially, but moving the `accept` into your loop. However, have your program read only a single input line from each connection. It should then close the connection and get ready to accept another.

If you haven't used Python much, you may need some help from a Python guru in getting the loop structure correct.

Right now, you'll probably be typing `^C` to stop your server. Python gurus should consider adding a `KeyboardInterrupt` handler to close the server socket when this happens.

Download the second server tester and try it out. This program makes a series of one-line connections to a server. It works best if more that one person is running it at a time. It will need modification.

Go ahead a modify your server program so that it will "read" a client connection until it is closed. This will mean having nested loops – an outer loop to accept connections and an inner loop to read from the connection. Download another server tester to test your improved server. Study the differences between the two server tester.

There are some rather serious problems with your present server. The first should be rather obvious. It can only handle one connection at a time. A web server with this property would be a disaster.

The second problem is a bit harder to see. Write a rather odd client that sends the number 2010 to the client, but with a 0.1 second delay between the 20 and the 10, something like:

```
s.send('20') ;
time.sleep(0.1) ;
s.send('10\n\r') ;
```

There's a good chance your server might have read 20 rather than 2010. (You might try to justify this because of the tenth of a second, but in embedded systems delays like this can be unintended. For example, a good Arduino Ethernet library always sends one character at a time which considerable delays between each character.)

Third attempt – Asynchronous I/O

This one is going to be hard. Look at section 5 of Gordon McMillian's Socket Programming HOWTO.

It might be a little hard for you to figure out this one on your own, so go ahead a download my third server, which uses the dreaded `select` routine. Modify the server so that it acts a bit like your present second server. This server isn't perfect. It does illustrate how to handle several simultaneous connections, but it won't handle input "lines" where there are delays between transmitted characters. Think of ways in which that problem could be solved.

Fourth attempt – Object-oriented server

Yes, you can write an object-oriented server.

UDP (User Datagram Protocol)
- Unreliable and session-less
- Servers will allocate ports
- Berkeley socket interface
  - Servers – socket, bind, recvfrom/sendto
  - Clients – socket, sendto/recvfrom

Exercise
- This time write a datagram-based program, you'll need both client and servers
  - Work in pairs if you wish

NAT (Network Address Translation)
- Technically, PAT – Port Address Translation
- The reason IPv4 is still going strong
- The two sides of the router
  - The Internet, with its allocated addresses
  - The local network, with its private addresses
- TCP session – Internal to external
  - Cast
    - Internal client (192.168.0.100, 5000)
    - External server (152.18.69.40, 80)
    - Router (75.100.110.120)
  - Actions
    - Router allocates (if necessary) translation for this "session"
      - Says port number 23456
      - [(192.168.0.100, 5000), (152.18.69.40, 80)] ↔ 23456
    - Connects to (152.18.69.40, 80) as (75.100.110.120, 23456)
    - When replies come from (152.18.69.40, 80) to 23456
    - Sent to (192.168.0.100, 5000)
- TCP session – External to internal
  - Generally requires configuration by the router manager

“Virtual server” ports are directed to specific internal computers

- UDP "session"
  - Could work almost the same
    - For session (192.168.0.100, 5000) ↔ (152.18.69.40, 53)],
      - Use UDP port number 23456
      - [(192.168.0.100, 5000), (152.18.69.40, 53)] ↔ 23456
  - Problems
    - UDP server may use a different port to respond
      - This allow server to use different processes for each "session"
    - In UDP peer-to-peer (gaming) applications
      - Several "internal" hosts will "broadcast" to a common port
      - XBox LIVE – open, moderate, and strict NAT

Firewalls

- Routers that filter traffic according to
  - Port numbers
  - Machine numbers
  - Packet content (viruses...)
  - All sorts of combinations of the above
- Generations (sort-of)
  - Packet filters – Steve Bellovin
  - Application layer firewall
  - Stateful firewall
- "Personal" firewall
  - `iptables` on penrose.cs.unca.edu

Exercise

- Set up a router (as best we can)

HTTP (Hypertext Transfer Protocol)

- **Not HTML**

SMTP (Simple Mail Transport Protocol)

SSH (Secure Shell)

SNMP (Simple Network Management Protocol)