Socket Programming

K. Yelamarthi & F. Walsh
Central Michigan University
Mt Pleasant, MI

Sockets Interface

- Important to communicate on the network
- Programming interface to perform network communication
- Can be used in many languages
- Based on client/server programming model

Sockets on a Client

Creating a generic network client:

- Create a socket
- Connect socket to server
- Send some data (a request)
- Receive some data (a response)
- Close the socket

Could repeat or stop based on the application

Create a Socket

```
import socket
mysock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

- Need to import the **socket** package
- **socket.socket()** creates the socket
- **AF_INET** declares the address family internet (IPv4)
  - AF_INET6 🖂 IPv6
  - AF_IRDA 🖂 Infrared
  - AF_BTH 🖂 Bluetooth
  - AF_APPLETALK 🖂 Apple Talk

- More information at https://msdn.microsoft.com/en-us/library/windows/desktop/ms740506(v=vs.85).aspx

Create a Socket

```
import socket
mysock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

- **SOCK_STREAM** indicates that we are using TCP (connection-based)
  - TCP has more overhead, but is reliable communication method
- SOCK_DGRAM supports datagrams, which are connectionless, unreliable buffers of fixed max length

Connect Socket to Server

```
host = socket.gethostbyname(“www.google.com”)
        or
host = ‘IP Address of server’
mysock.connect(host, 80)
```

- Need a host to connect to
- Host is an IP address, but you may only have the domain
- **gethostbyname()** performs DNS lookup
- **connect()** creates the connection
- Port is second argument, 80 is web traffic

Sending Data on a Socket

```
message = “GET / HTTP/1.1\ r\ n\ r \ n”
mysock.sendall(message)
```

- Message string is an HTTP GET request
  - Could send any data
- **\r\n** ⮽ carriage returns and line feeds

- **sendall()** ⮽ sends the data and tries until it succeeds

Receiving Data on a Socket

```
data = mysock.recv(1000)
```

- **recv()** returns data on the socket
  - Blocking wait, by default (it will sit there until it receives the response)
- Argument is the maximum number of bytes to receive
- Buffer size is optional in Python, but required in other languages such as C

```
mysock.close()
```

- Closes the socket

Sockets on the Server

Server needs to wait for requests

- Create a socket
- **Bind** the socket to an IP address and port
- **Listen** for a connection
- **Accept** the connection
- Receive the request
- Send the response

Creating and Binding a Socket

```
mysock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
mysock.bind(“”, 80)
```

- **bind()** binds the socket to a port
- First argument **“”** is the host, it is empty
  - Can receive from any host
- **80** refers to HTTP port, but can be changed to any other port

Listening and Accepting a Connection

```
mysock.listen(5)
conn, addr=mysock.accept()
```

- **listen()** starts listening for a **connect()**
  - Argument is **backlog**, number of requests allowed to wait for service
- **accept()** accepts a connection request
  - Returns a connection (for sending/receiving), and an address (IP, port)

Sending - Client

```
import socket
count=0
HOST = 'IP address of server'     # The remote host
PORT = 50007                      # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
while count < 1000:
    s.sendall('Hello, world')
    data = s.recv(1024)
    print data
    count += 1
s.close
```

Receiving - Server

```
import socket
HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
     print data
    if not data: break
    conn.sendall(data)
conn.close()
```

Task 1

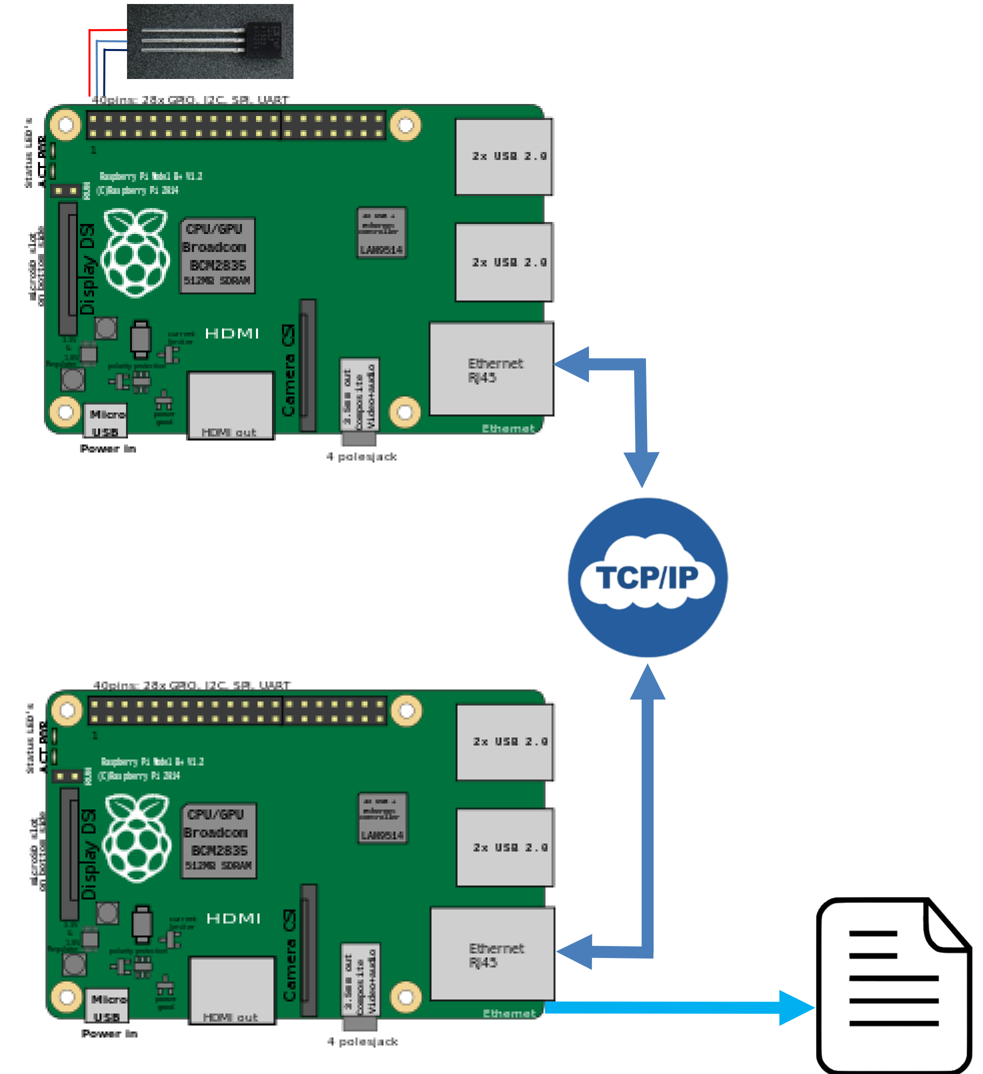
- Connect the temperature sensor (TMP36) to the Raspberry Pi to read temperature at a rate of 100 Hz

- Create a socket on the first Raspberry Pi and set it as a client

- Create a socket on the second Raspberry Pi and set it as a server

- Send the temperature sensor values from the client to server via a socket interface

- Store the first 100 temperature values obtained in a text file on the Server



Task 2

- Install tshark on the Raspberry Pi you are using as the server using the following commands:

```
sudo apt-get update
sudo apt-get install tshark
```

  Select Yes when prompted to allow non super users to capture data.

- Find the name of the network interface you are using on the pi (use the **ifconfig** command). It will probably be **eth0** if you are using ethernet or **wlan0** if using wifi. Use the following command to capture tcp traffic on the server port and write the data to file /tcp-capture.pcap. Update the interface if necessary:

```
sudo tshark -w /tcp-capture.pcap -i wlan0 -f "tcp port 50007"
```

- Now run the server.py and client.py tcp program again. Once finished, examine the tcp data recorded in /tcp-capture.pcap using the command:

```
sudo tshark -r /tcp-capture.pcap
```

Task 2(continued)

Examine the tcp data in the /tcp-capture.pcap file and answer the following questions (use the web where necessary):

- Locate the initial TCP 3-way handshake? Why is this necessary?
- ACK indicates an acknowledgment and PSH indicates a data push. What is the significance of the acknowledgments?
-

Each frame is allocated a number in the /tcp-capture.pcap file. Pick a frame number associated with a data push(PSH) from the leftmost column and examine it in more detail, for example:

```
tshark -r /tcp.traffic.pcap -V -Y "frame.number==12"
```

- Locate the data/payload(close to the end). Can you relate this to what was sent by the client? Do you think the data is "safe" during transmission?

```
Transmission Control Protocol, Src Port: 10500, Dst Port: 37075, Seq: 11, Ack: 25, Len: 14
    Source Port: 10500
    Destination Port: 37075
    [Stream index: 0]
    [TCP Segment Len: 14]
    Sequence number: 11    (relative sequence number)
    [Next sequence number: 25    (relative sequence number)]
    Acknowledgment number: 25    (relative ack number)
    Header Length: 32 bytes
    Flags: 0x018 (PSH, ACK)
        000. .... .... = Reserved: Not set
        ...0 .... .... = Nonce: Not set
        .... 0... .... = Congestion Window Reduced (CWR): Not set
        .... .0.. .... = ECN-Echo: Not set
        .... ..0. .... = Urgent: Not set
        .... ...1 .... = Acknowledgment: Set
        .... .... 1... = Push: Set
        .... .... .0.. = Reset: Not set
        .... .... ..0. = Syn: Not set
        .... .... ...0 = Fin: Not set
        [TCP Flags: ·······AP···]
. . . . . . .
Data (14 bytes)

0000  72 65 61 6c 6c 79 20 72 65 61 6c 6c 79 0a        really really.
    Data: 7265616c6c79207265616c6c790a
    [Length: 14]
```

Report

Include the following in your lab report

- If possible, include screen shots of your service working.
- Lessons learned
- Issues encountered and how you resolved them.
- Final version of the code you created (or a link to an online repository).
- Try to answer any questions in the lab.