CS 372 LECTURE 14

* The Transport Layer
  + Introduction
  + Multiplexing
* Transport services and protocols
  + Transport protocols
    - provide logical communication between application processes running on different hosts
    - run on end systems
    - Take communication from application layer protocol and make sure it can communicate with other application layer protocol correctly
* Transport services and protocols
  + sender protocol
    - accepts messages from application socket
    - breaks messages into data blocks
    - encapsulates blocks/ports into **segments**
    - passes segments/addresses to network layer
  + receiver protocol
    - accepts segments from network layer
      * strips ports
      * and decides which application
    - re-assembles data blocks into messages
    - passes messages to sockets at application layer
* Transport Later requires some Network Layer services
  + transport layer: logical communication between processes
  + Network layer: logical communication between hosts
    - packet addressing
    - route computation
    - packet forwarding
    - more later on the network layer….
* Multiplexing/demultiplexing
  + Multiplexing at sending hosts:
    - gathering data from multiple sockets, creating segments, encapsulating segments with header (later used for demultiplexing)
  + Demultiplexing at receiving host:
    - delivering received segments to correct socket
* How demultiplexing works
  + Host receives IP datagrams
    - each datagram has source IP address, destination IP address
    - each datagram encapsulates one transport-layer segment
    - each segment has source, destination port number
  + Host uses **IP addresses & port numbers** to direct segment to appropriate socket
* Connectionless demultiplexing
  + UDP socket identified by **destination (IP address, port number)**
    - **socket is a record that includes additional info**
  + When host receives UDP segment:
    - checks destination port number in segment
    - directs UDP segment to socket with that port number
  + IP datagrams with different source IP addresses and/or source port numbers might be directed to same socket
    - Like DNS
    - Multithreading is normally the answer
    - Receiving application has to set up different sockets
* Connection-oriented demultiplexing
  + TCP socket identified by
    - **source IP address**
    - **source port number**
    - **destination IP address**
    - **destination port number**
      * So receiving host can maintain the connection
  + receiving host uses all four values to direct segment to appropriate socket
  + server host may support many simultaneous TCP sockets
    - each socket identified by its own 4-tuple
  + web servers have different sockets for each connecting client
    - non-persistent HTTP will have different socket for each request
* Demultiplexing: example
  + diagram
  + Three segments all destined to IP address: B, dest port: 80 are demultiplexed to different sockets
* SUMMARY LECTURE 14
  + Definitions
    - segment
    - multiplexing
    - demultiplexing
  + Transport layer
    - inputs, outputs
    - responsibilities
    - sockets

CS 372 LECTURE #15

* Socket programming
  + Writing application layer protocols
  + sockets API
    - Application Programmers Interface
* Socket
  + OS-controlled interface (“a door”)
  + A logical port (implemented in software)
  + Created by and associated with an application on local host
  + An application process uses a socket to send/receive messages to/from another application process
* Socket Programming
  + Two socket types for two transport services:
    - UDP: unreliable datagram
    - TCP: reliable, byte stream
  + Application Example
    - 1. Client reads a line of characters (data) from its keyboard and send the data to the server.
    - 2. The server receives the data and converts characters to uppercase
    - 3. The server sends the modified data to the client
    - 4. The client receives the modified data and displays the line on its screen.
* Socket programming with UDP
  + UDP: no “connection” between client & server
    - no handshaking before sending data
    - **sender** explicitly attaches IP destination address and port # to each packet
    - **Receiver** extracts sender IP address and port # from received packet
  + UDP: transmitted data may be lost or received out-of-order
  + **Application viewpoint**: UDP provides unreliable transfer of groups of bytes (“datagrams”) between client and server
* Example application: UDP client
  + include python’s socket library
  + create UDP socket for server
  + get user keyboard input - message
  + attach server name, port to message, send into socket
  + read reply characters from socket into string – receive message up to 2048 characters
    - Send uses sendto
    - receive uses recvfrom
  + print out received string and close socket
    - Python UDPClient

from socket import \*

serverName = ‘hostname’

serverPort = 12000

clientSocket = socket(socket.AF\_INET, socket.SOCK\_DGRAM)

message = raw\_input(‘Input lowercase sentence:’)

clientSocket.sendto(message,(serverName, serverPort))

modifiedMessage, serverAddress = clientSocket.recvfrom(2048)

print modifiedMessage

clientSocket.close()

* SOCK\_DGRAM specifies the UDP client
* OS decides what the clients port number will be
* Example application: **UDP server** side of same program
  + Create UDP socket – SOCK\_DGRAM
  + bind socket to local port number 12000 – bind socket to port number that we chose
  + loop forever – server will run forever (while 1 in python)
  + Read from UDP socket into message, getting clients address (client IP and port)
  + send upper case string back to this client
    - Server converts to capital letters then send back
    - Python UDP sever

from socket import \*

serverPort = 12000

serverSocket = socket(AF\_INET, SOCK\_DGRAM)

serverSocket.bind((“,serverPort))

print “The server is ready to receive”)

while 1:

message, clientAddress = serverSocket.recvfrom(2048)

modifiedMessage = message.upper()

serverSocket.sendto(modifiedMessage, clientAddress)

* Socket Programming with TCP
  + **client must contact server**
    - server process must first be running
    - server must have created socket (door) that welcomes client’s contact
  + **client contacts server by:**
    - creating TCP socket, specifying IP address, port number of server process
    - **When client creates socket:**
      * client TCP establishes connection to server TCP
    - when contacted by client
      * **server TCP creates new socket** for server process to communicate with that particular client
        + allows server to talk with multiple clients
        + source port numbers used to distinguish clients
  + Application viewpoint:
    - TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server
* Example application: TCP client
  + create TCP socket for server, remote port 12000
    - uses SOCK\_STREAM instead of SOCK\_DGRAM
  + No need to attach server name, port
    - Socket will take care of getting it to the right place because the connection is there
  + Uses send and recv rather than sendto and recvfrom as UDP does
    - Python TCP client

from socket import \*

serverName = ‘servername’

serverPort = 12000

clientSocket = socket(AF\_INET, SOCK\_STREAM)

message = raw\_input(‘Input lowercase sentence:’)

clientSocket.connect((serverName, serverPort))

sentence = raw\_input(“Input lowercase sentence:’)

clientSocket.send(sentence)

modifiedSentence = clientSocket.recv(1024)

print ‘From Server:’, modifiedSentence

clientSocket.close()

* Example application: TCP server
  + create TCP welcoming socket
    - SOCK\_STREAM
  + server begins listening for incoming TCP requests
    - .listen
  + loop forever
  + server waits on accept() for incoming requests, new socket created on return
  + read bytes from socket (but noaddress as is UDP)
  + close connection to this client (but *not* welcoming socket)
    - Python TCPServer

from socket import \*

serverPort = 12000

serverSocket = socket(AF\_INET, SOCK\_STREAM)

serverSocket.bind((“,serverPort))

serverSocket.listen(1)

print “The server is ready to receive”

while 1:

connectionSocket, addr = serverSocket.accept()

sentence = connectionSocket.recv(1024)

capitalizedSentence = sentence.upper()

connectionSocket.send(capitalizedSentence)

connectionSocket.close()

* Project 1
  + see definition on course website
  + Programming using Socket API
    - implemented in C or python or C++ or Java (learn the most using C)
      * see references in the project description
    - **well-modularized and well-documented**
    - run on an OSU engr server
      * specify your testing machine in the program docs
    - don’t hard code any directories, since they might be inaccessible to the graders
    - cite any references and credit any collaborators
* Summary Lecture #15
  + Transport Layer
    - UDP
    - TCP
  + Socket programming
    - SOCK\_DGRAM for UDP
      * bind, sendto, recvfrom, close
    - SOCK\_STREAM for TCP
      * connect, bind, listen, accept, send, recv, close

CS 372 Lecture #16

* Reliable data transfer
  + motivations, concerns and principles
  + error detection
* Two Generals Problem
  + Cannot guarantee that message will be received
  + Cannot guarantee that received message has no errors
  + Is reliable messaging possible?
    - see “Two Generals problem” on Wikipedia
    - see also RFC1149
* Principles of reliable data transfer
  + implemented in application, transport, network, link layers
  + top-10 list of important networking topics!
  + characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
  + RELIABILTY OF A CHANNEL will be dependent of the reliability of the layers below it
* Reliable data transfer: getting started
  + 1. rdt\_send(): called from application layer. Data to be delivered to receive- side application layer
  + 2. udt\_send(): called by rdt, to transfer packet over reliable channel to receiver
  + 3. rdt\_rcv(): called from “below” when packet arrives on receive-side of channel
  + 4. deliver\_data(): called by rdt to deliver data to application layer
* Reliable data transfer: getting started
  + “reliable” is a relative term
  + small steps:
    - error detection
    - acknowledgement
    - sequencing
    - timing (flow/congestion control)
    - retransmission
    - fairness
  + Add reliability by adding reliable feature as those above
  + see textbook for development of reliable data transfer (using finite state diagrams)
    - **VERY IMPORTANT TO READ THIS SECTION OF TEXTBOOK**
* Error detection in UDP
  + In addition to port numbers, UDP segment header includes
    - 16-bit length field
    - 16-bit checksum field
      * used to detect errors where a bit has been changed or part of a sender has been lost
* UDP checksum
  + Goal: detect errors (e.g. flipped/lost bits) in transmitted segment
  + sender:
    - start checksum = 0
    - compute checksum
      * ones-complement of sim of segment content as 16-bit integers
      * see: [www.netfor2.com/checksum.html](http://www.netfor2.com/checksum.html)
        + why use ones complement rather than sum?
  + receiver:
    - compute checksum of received segment
    - compare computed checksum to segment checksum field
      * equal- no error detected
        + but may be errors anyway
      * not equal – error detected
        + discard entire packet
* Internet checksum: example
  + example: add two 16-bit integers
  + Note: when adding numbers, a carryout from the most significant bit needs to be added to the result
  + Checksum is the one’s complement of the sum
* UDP: Summary
  + “no frills,” “bare bones” transport protocol
  + “best effort” service
  + **basic error detection**
  + UDP segments may be
    - lost
    - delivered out-of-order
  + connectionless
    - no shaking between UDP sender, receiver
    - each UDP segment handled independently of others
  + UDP use:
    - streaming multimedia apps (loss tolerant, rate sensitive)
    - DNS
      * needs to be fast and there is too much traffic
  + **Why is there UDP?**
    - VERY FAST
    - no connection establishment (which can add delay)
    - simple: no connection state at sender, receiver
    - small header size
    - no congestion control, UDP can blast away as fast as desired
* SUMMARY LECTURE 16
  + Two general’s problem
  + Reliable data transfer
  + Error detection
  + Characteristics of UDP
    - Discussion Topic: When UDP detects an error, the packet is discarded without warning to the sender. Discuss the advantages and disadvantages of implementing error-correction at the transport layer (eg Hamming codes)