Lecture 2: Network Protocols and Sockets Programming (TCP Client)

What is the Internet?

Last lecture we said... on the Internet

- data is parcelled into packets
- each packet carries a destination address
- each packet is routed independently
- packets can arrive out of order
- packets may not arrive at all

On top of this packet-switched network, the Internet provides two types of delivery service:

- connectionless (datagram, UDP, e.g., streaming media, games)
- connection oriented (byte stream, TCP, e.g., web, email)

What is the Internet?

Connection oriented service provides:

- end-to-end reliability (sender retransmits lost packets)
- in-sequence delivery (receiver buffers incoming packets until it can deliver them in order)

Some fundamental questions about packet-switched network:

- how does a router know which router to forward a packet to?
- how does a receiver know the correct ordering of packets?
- how does a sender know which packet is lost and must be retransmitted?

The answer to all of these rely on network protocols

Network Protocols

Network protocols – rules (“syntax” and “grammar”) governing communication between nodes (sender, router, or receiver)

- example protocols?

Protocols define the format, order of messages sent and received among network entities, and actions taken to transmit message, and on message received
**Internet Protocol Stack**

- **application protocol**: support network applications
  - HTTP, SMTP, FTP, etc.
- **transport protocol**: endhost-to-endhost data transfer
  - TCP, UDP
- **network protocol**: routing of datagrams from source to destination
  - IP, routing protocols
- **link layer protocol**: data transfer between neighboring network elements
  - Ethernet, WiFi
- **physical protocol**: getting bits “on the wire”

---

**Layering in the IP Protocols**

![Layered Protocol Diagram]

**Not all Network Elements “Speak” All Layers**

**Why Layering?**

Networks are complex! Many “pieces”:
- applications
- hosts
- routers
- links of various media

One way to deal with complex systems:
- **explicit structure** separates out the pieces
- **modularization** makes system easier to maintain and update
  - changing the implementation of a layer is transparent to the rest
  - change of implementation ≠ change of service definition!
Creating a Network Application

Example benefits of layering:
• programmers can write apps that run on different end systems and communicate over a network e.g., browser communicates with web server
• no software written for devices in network core network core devices do no function at app layer

This design allows for rapid app development

Client-Server Computing

Server:
• a process that manages access to a resource • process or machine?
• usually has a well-known, permanent IP address • waits for connection • can use server farm/cluster or cloud computing for scaling • how do server farms maintain a single IP address externally?

Email (SMTP) uses the client-server paradigm

Client:
• a process that needs access to a resource • initiates connection with server • may be intermittently connected • may have dynamic IP addresses • clients do not communicate directly with each other

Alternative(s) to client-server?

Sockets

Process sends/receives messages to/from its socket

Socket analogous to door
• sending process shoves messages out the door • sending process relies on transport infrastructure on the other side of the door to deliver message to the socket at the receiver process

Client-Server Computing

Email (SMTP) uses the client-server paradigm

Client:
• a process that needs access to a resource • initiates connection with server • may be intermittently connected • may have dynamic IP addresses • clients do not communicate directly with each other

Alternative(s) to client-server?
Sockets API

An Application Programmer Interface (API) to access the network

• set of function prototypes, data structures, and constants
• allows programmer to learn once, write anywhere
• greatly simplifies the job of application programmers

Addressing Socket

A server host may support many simultaneous application processes, each with one or more sockets

• web servers, for example, uses a different socket for each connecting client

When a packet arrives, how does the kernel know which socket to forward it to?
• by the host’s unique 32-bit IP address?
• is the IP address sufficient to identify a socket?

How Demultiplexing Works

Host receives IP packets
• each packet has source and destination IP addresses
• each packet carries 1 transport-layer segment
• each segment has source and destination port numbers

Host uses IP addresses & port numbers to direct segment to the appropriate socket

TCP/UDP segment format
Connection-oriented Demux

Socket identifier includes both the IP addresses and port numbers associated with the socket on the host.

Example port numbers:
- HTTP server: 80
- Mail server: 25
- See /etc/services

Receiver kernel uses all four values to direct packet to appropriate socket.

Socket Addresses

Somewhere in the socket structure:

- bind(): IP address, Port#
- connect(): IP address, Port#

TCP Server:

- IP address, Port#
- matched against incoming packet destination
- copied to outgoing packet destination

TCP Client:

- IP address, Port#
- client’s address, ephemeral
- server’s address, well-known

Sockets

What exactly are sockets?
- an endpoint of a connection
  - identified by the IP address and port number of both sender and receiver
- API similar to UNIX file I/O API (provides a file descriptor)

Berkeley sockets is the most popular network API
- runs on Linux, Mac OS X, Windows
- can build higher-level interfaces on top of sockets
  - e.g., Remote Procedure Call (RPC)

Based on C, single threaded model
- does not require multiple threads

Process File Table and Socket Descriptor

Stevens TCP/IP Illustrated v.1 p.446
Types of Sockets

Different types of sockets implement different service models
• data stream vs. datagram

Data stream socket (e.g., TCP)
• connection-oriented
  • reliable, in order delivery
  • at-most-once delivery, no duplicates
• used by e.g., smtp, http, ssh

Datagram socket (e.g., UDP)
• connectionless (just data-transfer)
  • “best-effort” delivery, possibly lower variance in delay
• used by e.g., IP telephony, streaming audio,
  streaming video, multi-player gaming, etc.

Data Stream vs. Datagram

Data stream treats data as one continuous stream, not chopped up into separate “chunks”

Simplified E-mail Delivery

You want to send email to friend@cs.usc.edu

At your end, your mailer (client)
• translates cs.usc.edu to its IP address (128.125.1.45)
• decides to use TCP as the transport protocol (Why?)
• creates a socket
• connects to 128.125.1.45 at the well-known SMTP port # (25)
• parcels out your email into packets
• sends the packets out

On the Internet, your packets got:
• transmitted
• routed
• buffered
• forwarded, or
• dropped

At the receiver, smtpd (server)
• must make a “receiver” ahead of time:
• creates a socket
• decides on TCP
• binds the socket to smtp’s well-known port #
• listens on the socket
• accepts your smtp connection requests
• receives your email packets
Stream/TCP Sockets

When a TCP server accepts a client, it returns a new socket to communicate with the client

- allows server to talk to multiple clients
- source address & port number used to distinguish clients

Initialize (TCP Client)

```c
int sd;
if ((sd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
    perror("socket");
    printf("Failed to create socket\n");
    abort();
}
```

socket() creates a socket data structure and attaches it to the process’s file descriptor table

Handling errors that occur rarely usually consumes most of systems code

Establish (TCP Client)

```c
unsigned short server_port;
char *servername;  // both assume initialized
struct sockaddr_in sin;
struct hostent *host = gethostbyname(servername);
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
in.sin_addr.s_addr = *(unsigned long *) host->h_addr_list[0];
in.sin_port = htons(server_port);
if (connect(sd, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
    perror("connect");
    printf("Cannot connect to server\n");
    abort();
}
```

connect() initiates connection (for TCP)

Sending Data Stream (TCP Client)

```c
int send_packets(char *buffer, int buffer_len)
{
    sent_bytes = send(sd, buffer, buffer_len, 0);
    if (send_bytes < 0)
        perror("send");
    return 0;
}
```

- returns how many bytes are actually sent
- must loop to make sure that all is sent (unless blocking I/O)

What is blocking and non-blocking I/O?
Why do you want to use non-blocking I/O?