CS 455/555 / Spring 2013
Intro to Networks and Communications

Client/Server Computing and Socket Programming

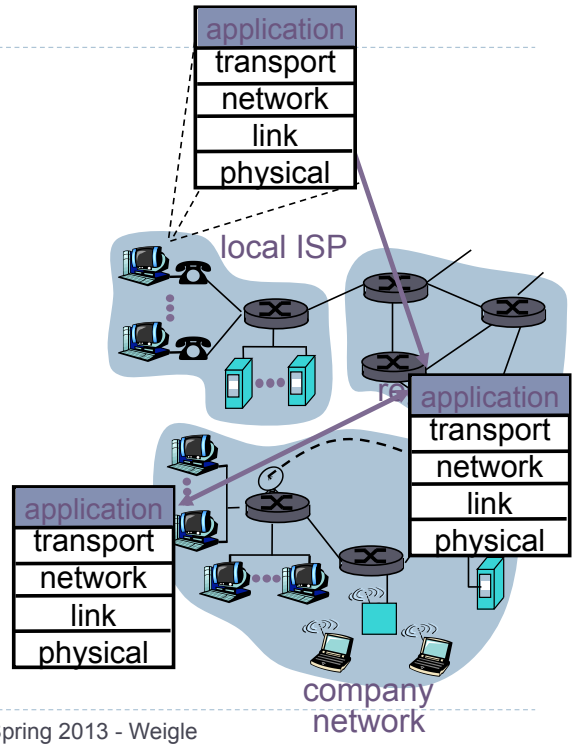
Dr. Michele C. Weigle

http://www.cs.odu.edu/~mweigle/CS455-S13/

Applications and Application-Layer Protocols

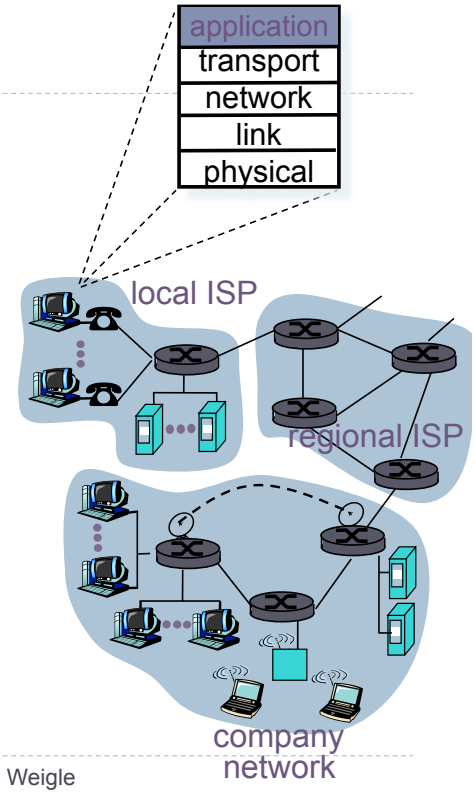
Overview

- Application: Communicating, distributed processes
  - Running in network hosts in "user space"
  - Exchange messages to implement application
- Application-layer protocols
  - One "piece" of an application
  - Defines messages exchanged and actions taken
  - Uses services provided by lower layer protocols



Application-Layer Protocols
Overview

- Application-layer protocols define:
  - The types of messages exchanged
  - The syntax and semantics of messages
  - The rules for when and how messages are sent
- Public protocols (defined in RFCs)
  - HTTP, FTP, SMTP, POP, IMAP, DNS
- Proprietary protocols
  - RealAudio, RealVideo
  - IP telephony
  - …



Network Working Group
Request for Comments: 2616
Obsoletes: 2068
Category: Standards Track

June 1999

R. Fielding UC Irvine
J. Gettys Compaq/W3C
J. Mogul Compaq
H. Frystyk W3C/MIT
L. Masinter Xerox
P. Leach Microsoft
T. Berners-Lee W3C/MIT

Hypertext Transfer Protocol -- HTTP/1.1

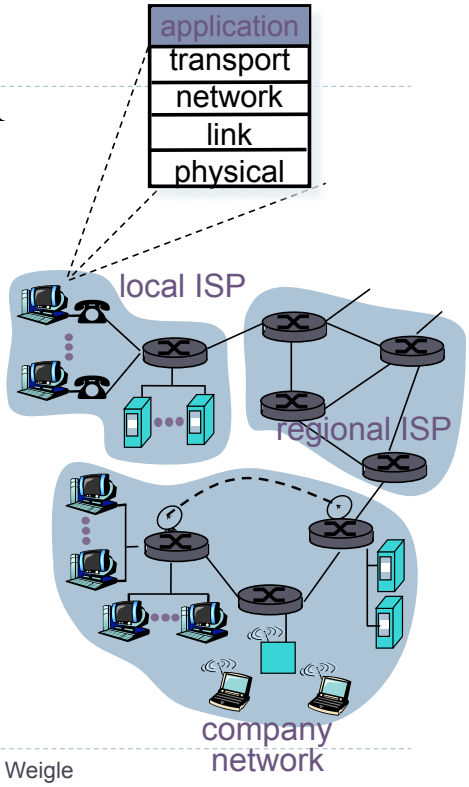
Abstract

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers [47]. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.

HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification defines the protocol referred to as "HTTP/1.1", and is an update to RFC 2068 [33].

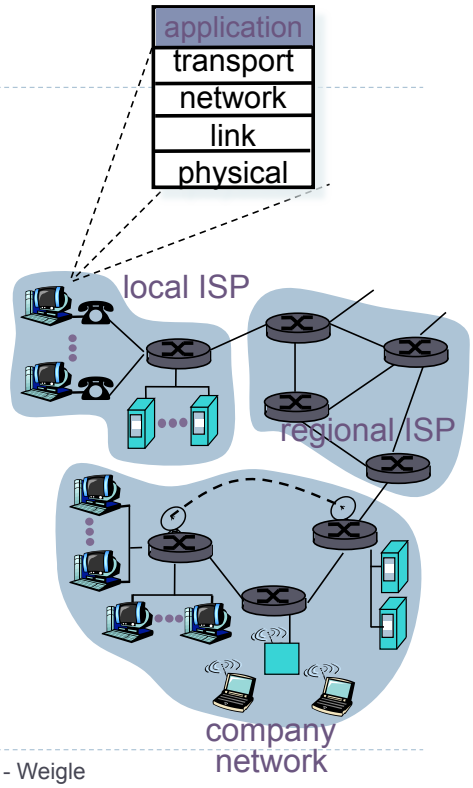
Application-Layer Protocols
Outline

- The architecture of distributed systems (KR 2.1)
  - Client/Server computing
  - P2P computing
  - Hybrid (Client/Server and P2P) systems
- Socket programming (KR 2.7-2.8)
  - programming model used in constructing distributed systems
- Example client/server systems and their application-level protocols
  - The World-Wide Web (HTTP) - KR 2.2
  - Reliable file transfer (FTP) - KR 2.3
  - E-mail (SMTP & POP) - KR 2.4
  - Internet Domain Name System (DNS) - KR 2.5

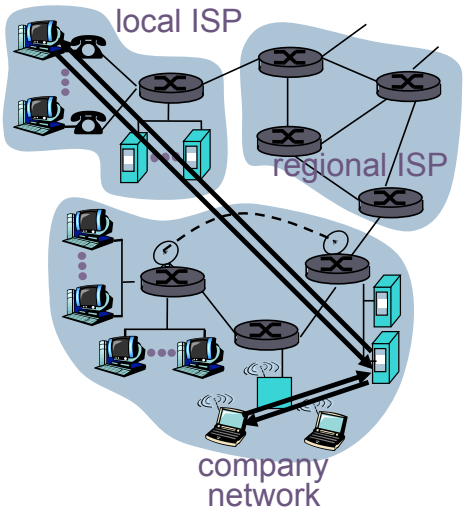


Application-Layer Protocols
Outline

- Protocol design issues:
  - In-band vs. out-of-band control signaling
  - Push vs. pull protocols
  - Persistent vs. non-persistent connections
- Client/server service architectures
  - Contacted server responds versus forwards request



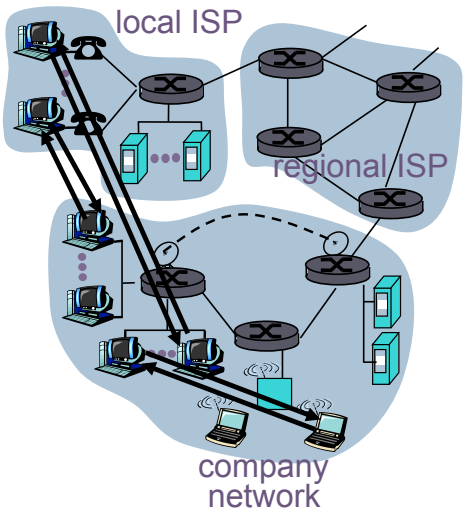
Application-Layer Protocols
Client-Server Architecture



- Server:
  - always-on host
  - permanent IP address
  - server farms for scaling
- Clients:
  - communicate with server
  - may be intermittently connected
  - may have dynamic IP addresses
  - do not communicate directly with each other

Application-Layer Protocols
Pure P2P Architecture

- No always-on server
- Arbitrary end systems directly communicate
- Peers are intermittently connected and change IP addresses
- Example: Gnutella
- Highly scalable
- But difficult to manage



Application-Layer Protocols
Hybrid of Client-Server and P2P

- Napster
  - File transfer P2P
  - File search centralized:
    - Peers register content at central server
    - Peers query same central server to locate content
- Instant messaging
  - Chatting between two users is P2P
  - Presence detection/location centralized:
    - User registers its IP address with central server when it comes online
    - User contacts central server to find IP addresses of buddies

Application-Layer Protocols
Transport Services

- Data loss
  - Some apps (e.g., audio) can tolerate some loss
  - Other apps (e.g., file transfer, telnet) require 100% reliable data transfer
- Timing
  - Some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"
- Bandwidth
  - Some apps (e.g., multimedia) require minimum amount of bandwidth to be "effective"
  - Other apps ("elastic apps") make use of whatever bandwidth they get

Internet Applications
Transport Service Requirements

Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video:10kbps-5Mbps	yes, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 100's msec
instant messaging	no loss	elastic	yes and no

Internet Transport Protocols
Services Provided

- TCP service:
  - *connection-oriented*: setup required between client, server
  - *reliable transport* between sending and receiving process
  - *flow control*: sender won't overwhelm receiver
  - *congestion control*: throttle sender when network overloaded
  - *does not provide*: timing, minimum bandwidth guarantees
- UDP service:
  - *unreliable* data transfer between sending and receiving process
  - *does not provide*: connection setup, reliability, flow control, congestion control, timing, or minimum bandwidth guarantees

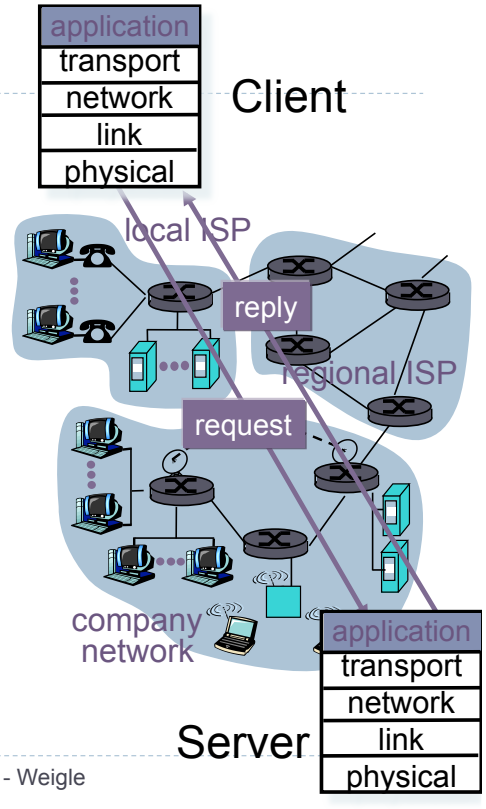
Why bother? Why is there a UDP?

Internet Applications
Application and Transport Protocols

Application	Application layer protocol	Underlying transport protocol
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	HTTP (YouTube), proprietary (RealNetworks)	TCP or UDP
Internet telephony	proprietary (e.g., Skype)	typically UDP

The Application Layer
The client-server paradigm

- Typical network application has two pieces: *client* and *server*
- Client:
  - Initiates contact with server ("speaks first")
  - Requests service from server
  - For Web, client is implemented in browser; for e-mail, in mail reader
- Server:
  - Provides requested service to client
  - "Always" running
  - May also include a "client interface"



Client/Server Paradigm
Socket programming

- Sockets are the fundamental building block for client/server systems
- Sockets are created and managed by applications
  - Strong analogies with files

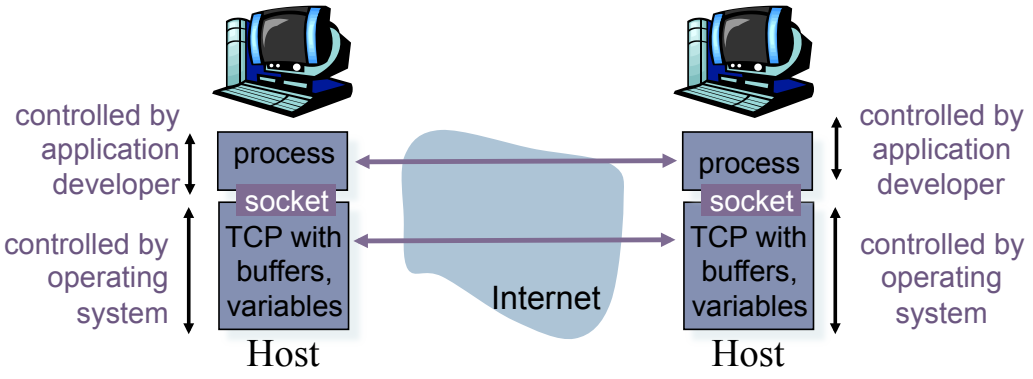
socket

a *host-local, application created/released, OS-controlled* interface into which an application process can *both send and receive* messages to/from another (remote or local) application process

- Two types of transport services are available via the socket API:
  - UDP sockets: unreliable, datagram-oriented communications
  - TCP sockets: reliable, stream-oriented communications

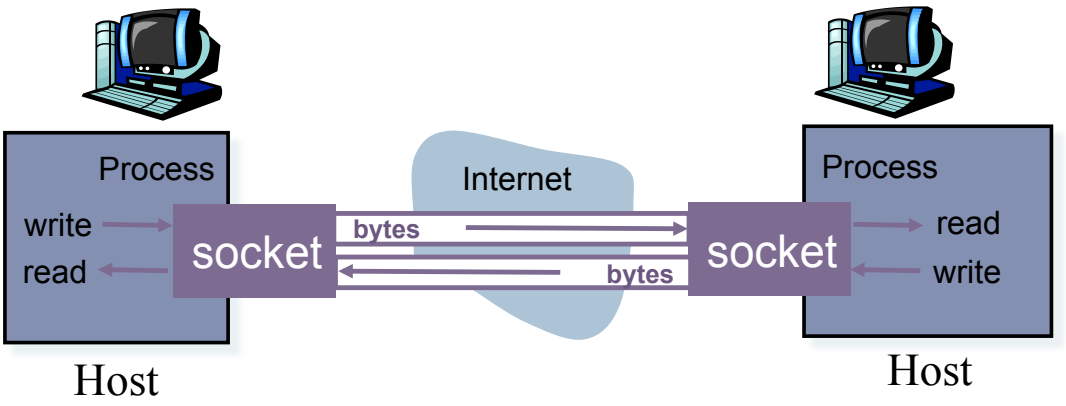
Client/Server Paradigm
Socket-programming using TCP

- A socket is an application created, OS-controlled interface into which an application can both send and receive messages to and from another application
  - A "door" between application processes and end-to-end transport protocols



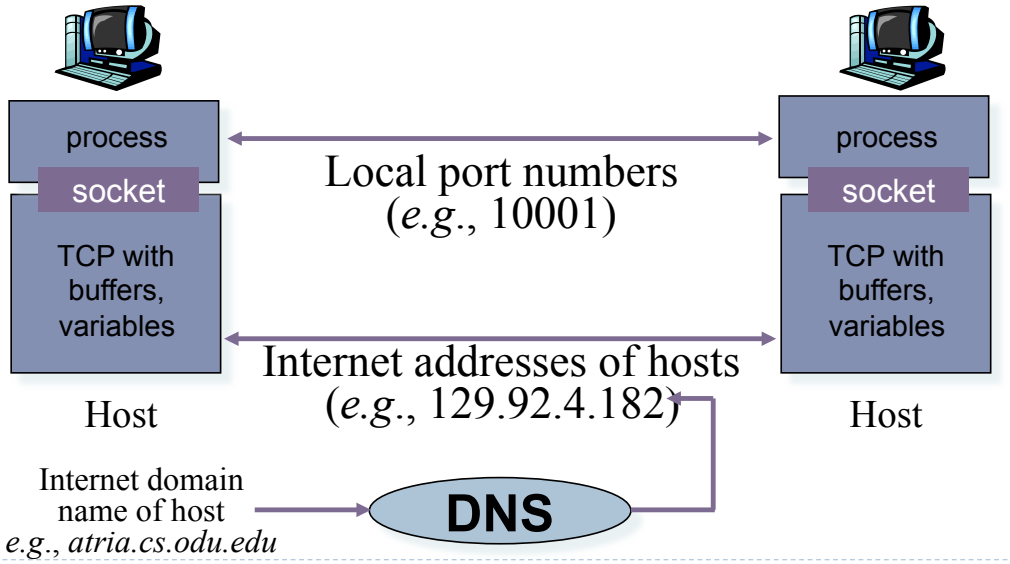
Socket Programming using TCP
TCP socket programming model

- A TCP socket provides a reliable bi-directional communications channel from one process to another
  - A "pair of pipes" abstraction



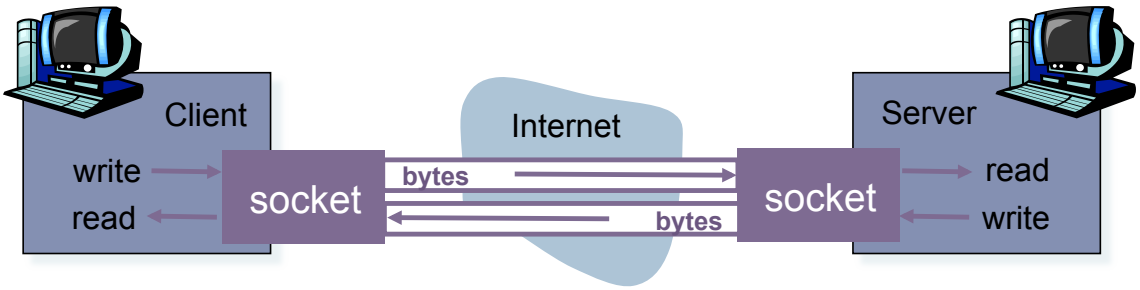
Socket Programming using TCP
Network addressing for sockets

- Sockets are addressed using an IP address and port number



Socket Programming using TCP

Socket programming in general

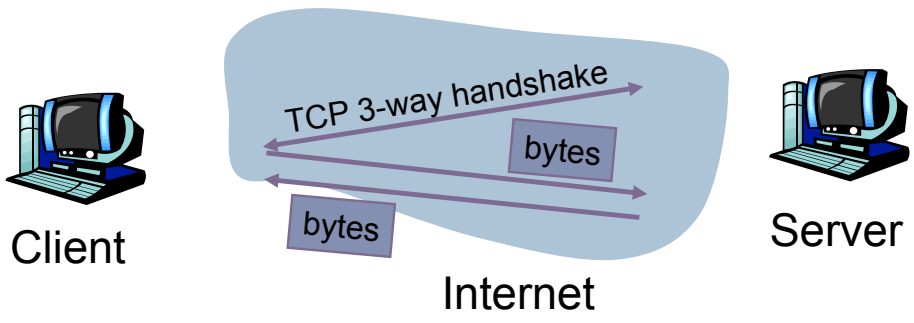


- Client creates a local TCP socket specifying the IP and port number of server process
  - if necessary, client resolves IP address from hostname
- Client contacts server
  - Server process must be running
  - Server must have created socket that "welcomes" client's contact
- When the client creates a socket, the client's TCP establishes connection to server's TCP
- When contacted by a client, server creates a new socket for server process to communicate with client
  - This allows the server to talk with multiple clients

Client-Server Model

TCP

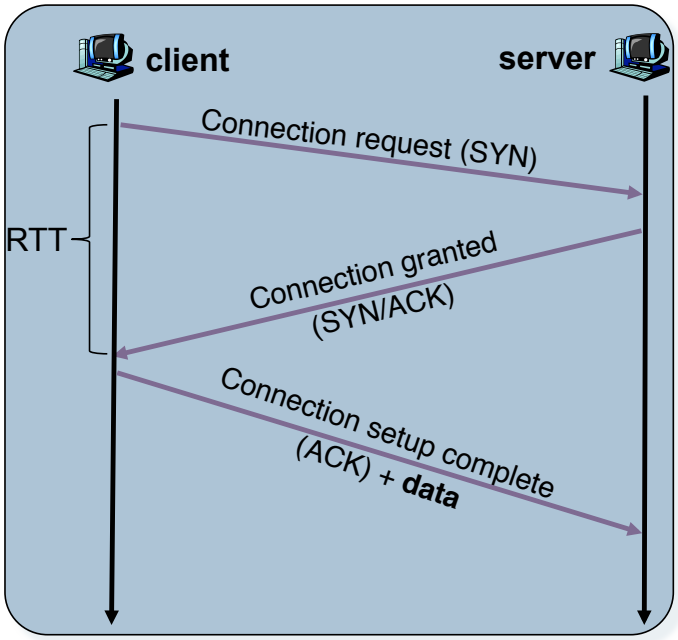
- Before data can be transmitted, TCP requires *connection setup*
  - called a 3-way handshake



Client-Server Model and TCP

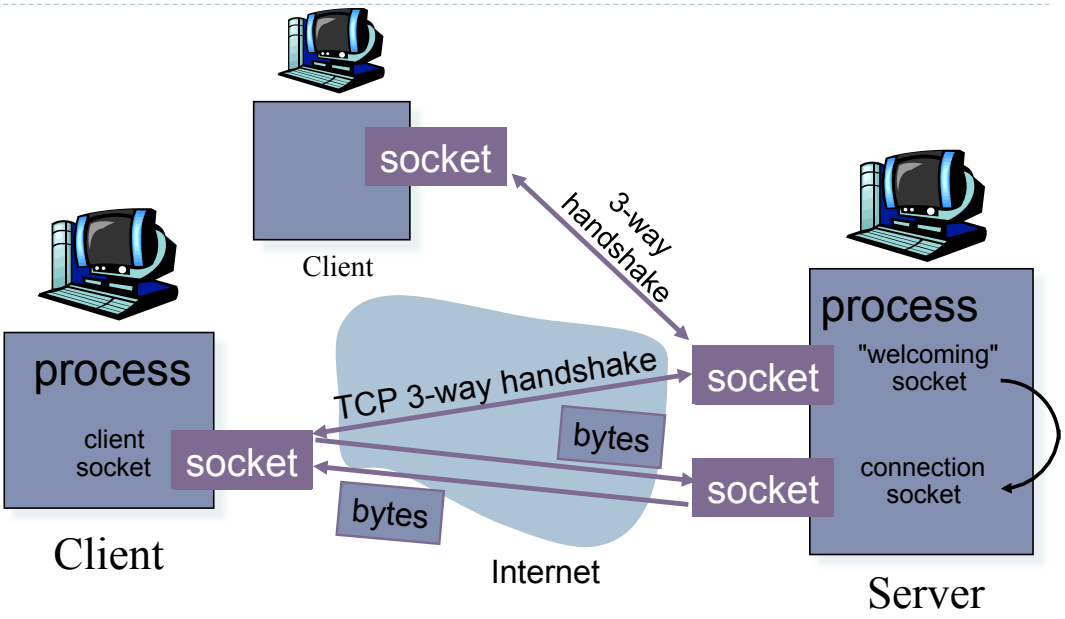
The three-way handshake

- Client sends SYN segment to server
- Server receives SYN, replies with SYN +ACK segment
  - ACKs received SYN
- Third segment may be an ACK only or an ACK+data
- Takes 1 round-trip time (RTT) to complete



Socket Programming using TCP

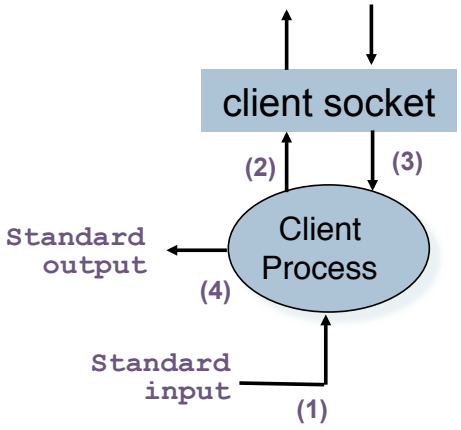
Socket creation in the client-server model



Socket Programming with TCP

Client Structure

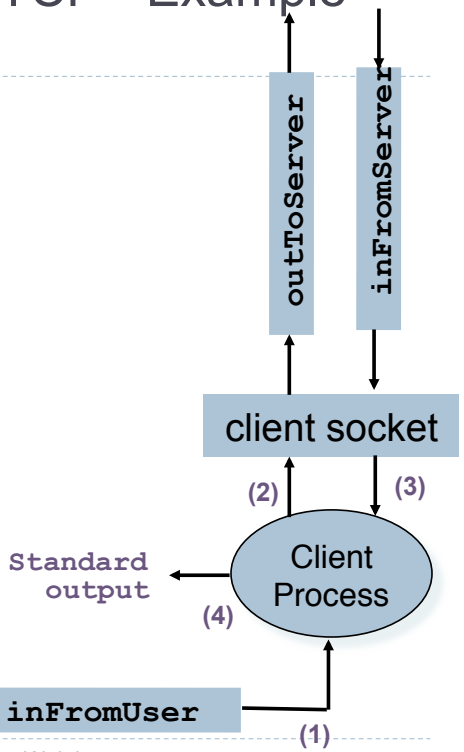
- Client reads from standard input, writes to server via a socket
- Server reads line from a socket
- Server converts line to uppercase and writes back to client
- Client reads from socket, prints modified line to standard output



Socket Programming with TCP - Example

Client Structure - Java

- Client reads from standard input (`inFromUser` stream), writes to server via a socket (`outToServer` stream)
- Server reads line from a socket
- Server converts line to uppercase and writes back to client
- Client reads from socket, (`inFromServer` stream) prints modified line to standard output



TCP Client
Examples - Java and Python

- Network code is similar
- Differences are in structure of code
  - Java - class-based, more complicated input/output
- TCPClient.py
- TCPClient.java

http://www.cs.odu.edu/~mweigle/CS455-S13/Code

Socket Programming with TCP
Server Structure

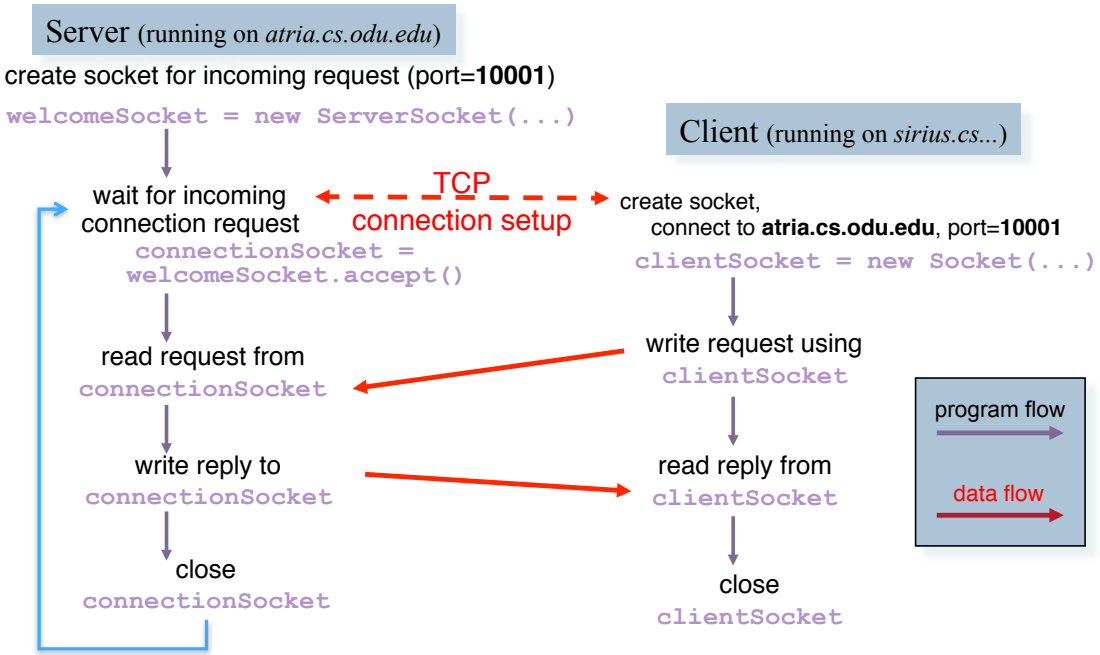
- Server listens on welcoming socket, waiting for connection from client
- Connecting client causes a new socket to be created on server
- Server reads input from socket
- Server converts line to uppercase and writes back to client
- Server closes connection socket

TCP Server
Examples - Java and Python

- As with client, network code is similar
- Differences are in structure of code
  - Java - class-based, more complicated input/output
- TCPServer.py
- TCPServer.java

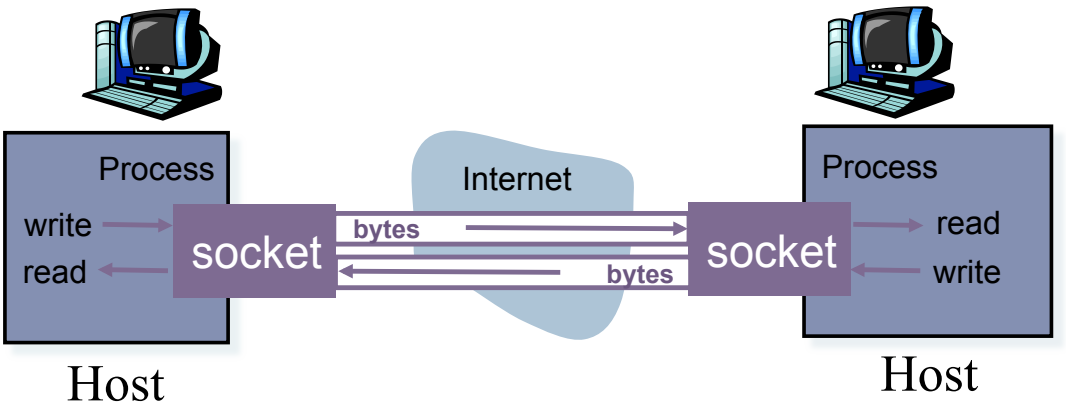
http://www.cs.odu.edu/~mweigle/CS455-S13/Code

Socket Programming with TCP - Example
Client/server TCP socket interaction in Java



Socket Programming using UDP
UDP socket programming model

- A UDP socket provides an unreliable bi-directional communication channel from one process to another
  - A "datagram" abstraction



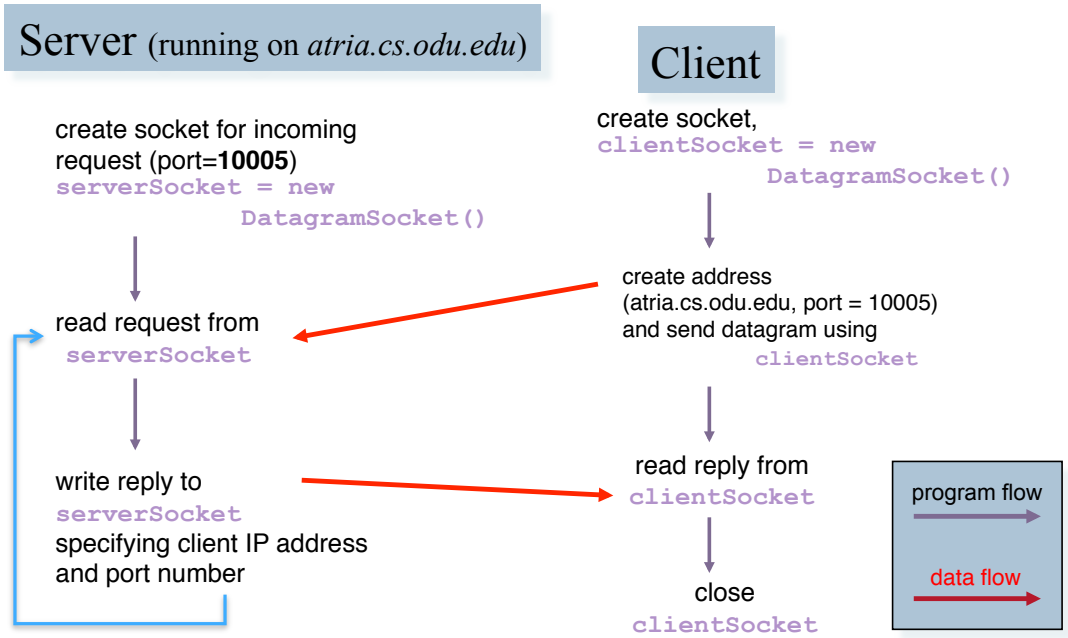
UDP Client
Examples - Java and Python

- Java network code makes the datagram nature of UDP explicit
  - must create datagram and fill it with data and receiver address
- With UDP, there is no listening socket. Client and server code are very similar.
- UDPClient.py, UDPClient.java
- UDPServer.py, UDPClient.java

http://www.cs.odu.edu/~mweigle/CS455-S13/Code

Socket Programming with UDP - Example

Client/server UDP socket interaction in Java



Socket Programming

Message Boundaries

- TCP does not preserve message boundaries
  - just a stream of bytes
  - one call to readLine() may return data from multiple packets
    - using BufferedReader in Java hides this
- UDP does preserve message boundaries
  - datagrams are separate entities
  - one call to receive() returns only a single datagram

Socket Programming
Out of Class Practice Exercises

- Modify the TCPClient so that it doesn't quit after receiving the reply (allow the user to send and receive multiple messages)
- Run one server and connect multiple clients to the same server to see how they are handled.
- Modify the TCPServer so that it prints the IP address and port of the connected client.
- Do the same for UDPClient and UDPServer
- Run the Java {TCP,UDP} client against the Python {TCP, UDP} server. Does it work?

Application-Layer Protocols
Outline

- The architecture of distributed systems (KR 2.1)
  - Client/Server computing
  - P2P computing
  - Hybrid (Client/Server and P2P) systems
- Socket programming (KR 2.7-2.8)
  - programming model used in constructing distributed systems
- Example client/server systems and their application-level protocols
  - The World-Wide Web (HTTP) - KR 2.2
  - Reliable file transfer (FTP) - KR 2.3
  - E-mail (SMTP & POP) - KR 2.4
  - Internet Domain Name System (DNS) - KR 2.5