CPSC 826
Internetworking

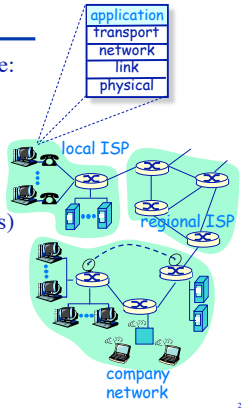
Client/Server Computing & Socket Programming

Michele Weigle
Department of Computer Science
Clemson University
mweigle@cs.clemson.edu
September 1, 2004

http://www.cs.clemson.edu/~mweigle/courses/cpsc826

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
W3C/MIT

June 1999
Microsoft

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

Hypertext Transfer Protocol -- HTTP/1.1

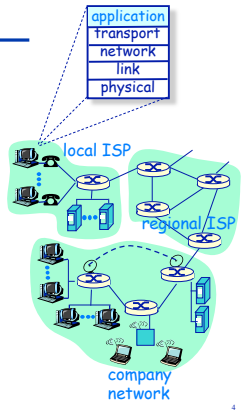
Abstract

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].

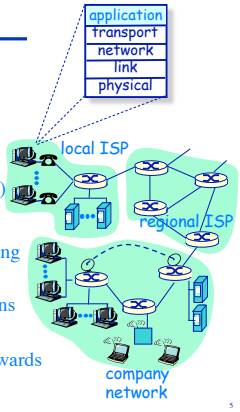
Application-Layer Protocols
Outline

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



Application-Layer Protocols
Outline

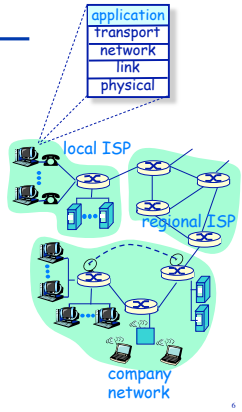
- Example client/server systems and their application-level protocols
  - » The World-Wide Web (HTTP)
  - » Reliable file transfer (FTP)
  - » E-mail (SMTP & POP)
  - » Internet Domain Name System (DNS)
- Protocol design issues:
  - » In-band *v.* out-of-band control signaling
  - » Push *v.* pull protocols
  - » Persistent *v.* non-persistent connections
- Client/server service architectures
  - » Contacted server responds versus forwards request



5

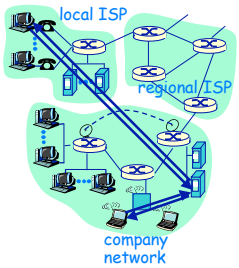
Application-Layer Protocols
Outline

- Example P2P systems and their application-level protocols
  - » Gnutella
  - » Napster (hybrid)
  - » KaZaA



6

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

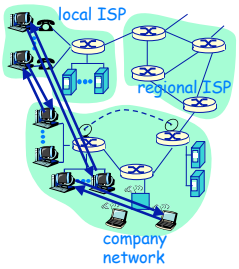
7

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



8

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

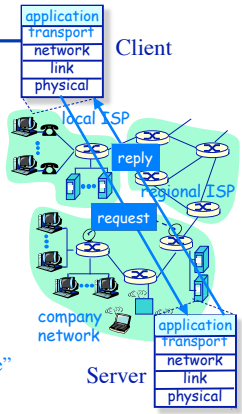
Why bother? Why is there a UDP?

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	proprietary (e.g. RealNetworks)	TCP or UDP
Internet telephony	proprietary (e.g., Dialpad)	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

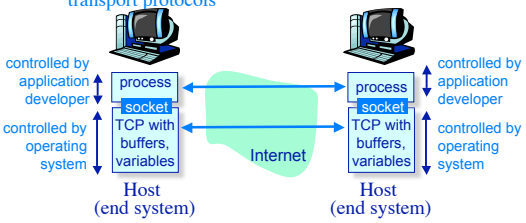
- Sockets are the fundamental building block for client/server systems
- Sockets are created and managed by applications
  - » Strong analogies with files
- Two types of transport services are available via the socket API:
  - » UDP sockets: unreliable, datagram-oriented communications
  - » TCP sockets: reliable, stream-oriented communications

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

Client/Server Paradigm
Socket-programming using TCP

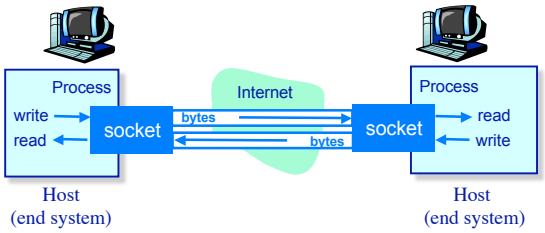
- A socket is an application created, OS-controlled interface into which an application can both send and receive messages to and from another application
  - » A "door" between application processes and end-to-end transport protocols



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

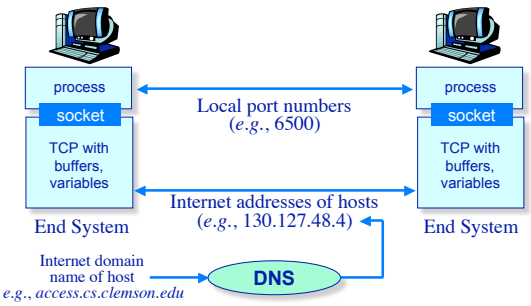


17

Socket-programming using TCP

Network addressing for sockets

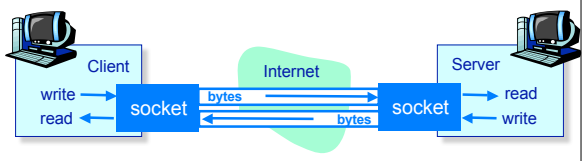
- Sockets are addressed using an IP address and port number



18

Socket-programming using TCP

Socket programming in Java

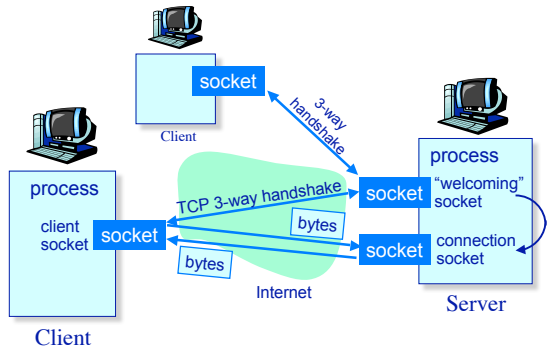


- Client creates a local TCP socket specifying the host and port number of server process
  - » Java resolves host names to IP addresses using DNS
- Client contacts server
  - » Server process must be running
  - » Server must have created socket that "welcomes" client's contact
- When the client creates 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

19

Socket-programming using TCP

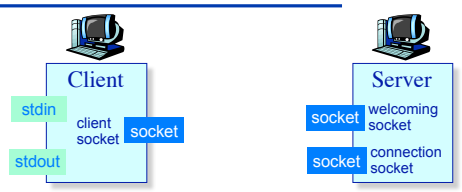
Socket creation in the client-server model



20

Socket-programming using TCP

Simple client-server example

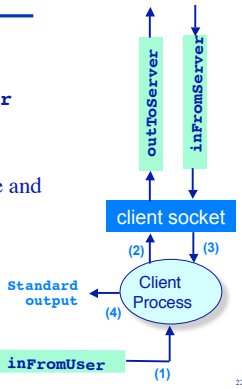


- The client reads a line of text from standard input and sends the text to the server via a socket
- The server receives the line of text from the client and converts the line of characters to all uppercase
- The server sends the converted line back to the client
- The client receives the converted text and writes it to standard output

Socket programming with TCP Example

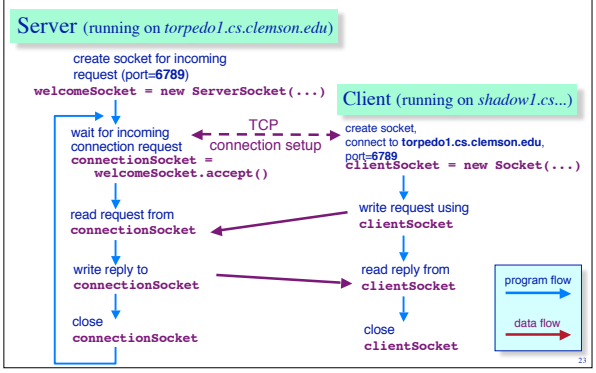
Client structure

- 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



Socket programming with TCP Example

Client/server TCP socket interaction in Java



Socket programming with TCP Example

Java client

```
import java.io.*;
import java.net.*;
class TCPClient {
    public static void main(String argv[]) throws Exception
    {
      String sentence;
      String modifiedSentence;

// Create (buffered) input stream using standard input
      BufferedReader inFromUser = new BufferedReader(
                                    new InputStreamReader(System.in));

      System.out.println("Client ready for input");


// Create client socket with connection to server at port 6789

        Socket clientSocket = new Socket("torpedo1.cs.clemson.edu", 6789);
```

Socket programming with TCP Example
Java client II

```
// Create output stream attached to socket
    DataOutputStream outToServer = new DataOutputStream(
                                        clientSocket.getOutputStream());

// Create (buffered) input stream attached to socket
    BufferedReader inFromServer = new BufferedReader(
                                    new InputStreamReader(
                                        clientSocket.getInputStream()));
// Write line to server
    outToServer.writeBytes(sentence + '\n');

// Read line from server
    modifiedSentence = inFromServer.readLine();

    System.out.println("FROM SERVER: " + modifiedSentence);
    clientSocket.close();

  } // end main
} // end class
```


Socket programming with TCP Example
Java server

```
import java.io.*;
import java.net.*;
class TCPServer {
  public static void main(String argv[]) throws Exception
    {
      String clientSentence;
      String capitalizedSentence;

// Create "welcoming" socket using port 6789
      ServerSocket welcomeSocket = new ServerSocket(6789);

      System.out.println("Server Ready for Connection");

// While loop to handle arbitrary sequence of clients making requests
      while(true) {

// Waits for some client to connect and creates new socket for connection
        Socket connectionSocket = welcomeSocket.accept();

        System.out.println("Client Made Connection");
```


Example
Java server II

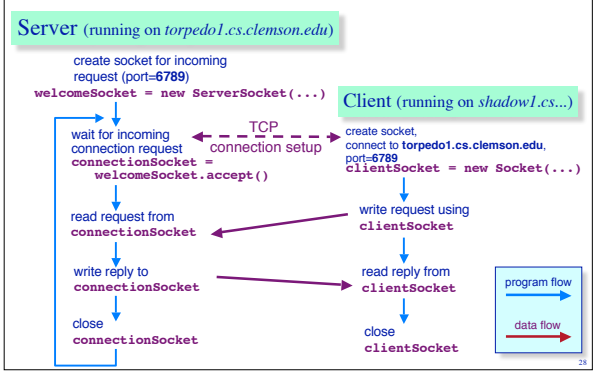
```
// Create (buffered) input stream attached to connection socket
      BufferedReader inFromClient = new BufferedReader(
                                    new InputStreamReader(
                                        connectionSocket.getInputStream()));
// Create output stream attached to connection socket
      DataOutputStream outToClient = new DataOutputStream(
                                        connectionSocket.getOutputStream());
// Read input line from socket
      clientSentence = inFromClient.readLine();

      System.out.println("Client sent: " + clientSentence);
      capitalizedSentence = clientSentence.toUpperCase() + '\n';

// Write output line to socket
      outToClient.writeBytes(capitalizedSentence);
      connectionSocket.close();

    } // end while; loop back to accept a new client connection
  } // end main
} // end class
```

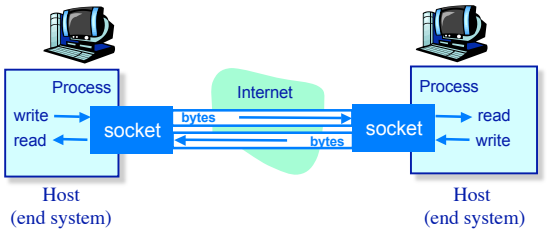

Socket programming with TCP Example
Client/server TCP socket interaction in Java



Socket-programming using UDP

UDP socket programming model

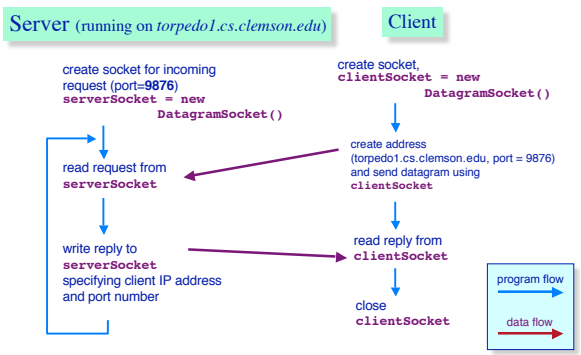
- A UDP socket provides an *unreliable* bi-directional communication channel from one process to another
  - » A "datagram" abstraction



29

Socket programming with UDP Example

Client/server UDP socket interaction in Java



30