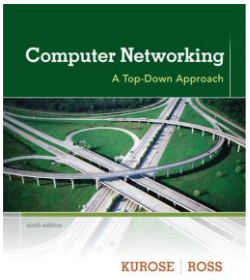
Chapter 2
Application Layer



Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

A note on the use of these ppt slides:
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Chapter 2: outline

2.1 Principles of network applications
2.2 Web and HTTP
~~2.3 FTP~~
2.4 electronic mail
- SMTP, ~~POP3, IMAP~~

2.5 DNS
2.6 P2P applications
2.7 socket programming with UDP and TCP

Chapter 2: application layer

our goals:

- conceptual, implementation aspects of network application protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-level protocols
  - HTTP
  - SMTP
  - DNS
- creating network applications
  - socket API

Some network apps

- e-mail
- web
- text messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)
- voice over IP (e.g., Skype)
- real-time video conferencing
- social networking
- search
- …
- …

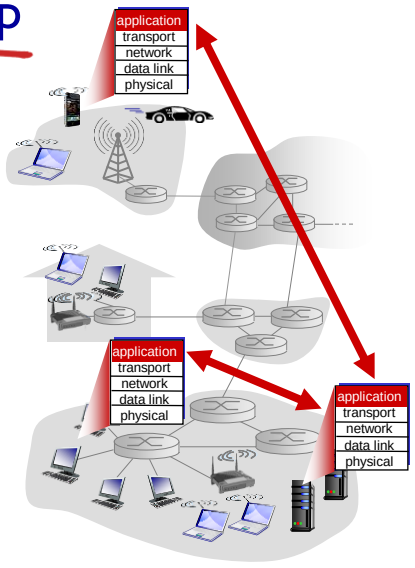
Creating a network app

write programs that:

- run on (different) *end systems*
- communicate over network
- e.g., web server software communicates with browser software

no need to write software for network-core devices

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation

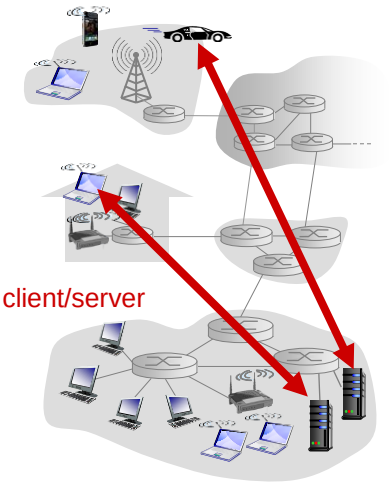


Application architectures

possible structure of applications:

- client-server
- peer-to-peer (P2P)

Client-server architecture



server:

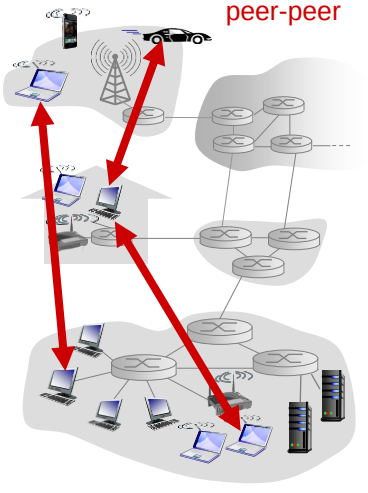
- always-on host
- permanent IP address
- data centers for scaling

clients:

- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other

P2P architecture

- *no* always-on server
  - Sometimes for discovery
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management



Processes communicating

process: program running within a host

- within same host, two processes communicate using inter-process communication (defined by OS)
- processes in different hosts communicate by exchanging messages

clients, servers

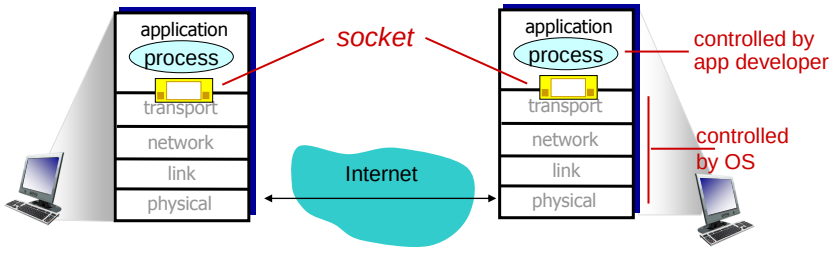
client process: process that initiates communication

server process: process that waits to be contacted

- aside: applications with P2P architectures have client processes & server processes

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- *Q:* does IP address of host on which process runs suffice for identifying the process?
  - *A:* no, *many* processes can be running on same host
- *identifier* includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80
- The first 1024 ports are "well-known" ports
- more shortly…

App-layer protocol defines

- types of messages exchanged,
  - e.g., request, response
- message syntax:
  - what fields in messages & how fields are delineated
- message semantics
  - meaning of information in fields
- rules for when and how processes send & respond to messages

open protocols:

- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype

What transport service does an app need?

data integrity

- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- other apps (e.g., audio) can tolerate some loss

timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

throughput

- some apps (e.g., multimedia) require minimum amount of throughput to be "effective"
- other apps ("elastic apps") make use of whatever throughput they get

security

- encryption, data integrity, …

Transport service requirements: common apps

application	data loss	throughput	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
text messaging	no loss	elastic	yes and no

Internet transport protocols services

TCP service:

- *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 throughput guarantee, security
- *connection-oriented:* setup required between client and server processes

UDP service:

- *unreliable data transfer* between sending and receiving process
- *does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, orconnection setup,

Q: why bother? Why is there a UDP?

Internet apps: application, 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 (e.g., YouTube), RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	TCP or UDP

Securing TCP

TCP & UDP

- no encryption
- cleartext passwds sent into socket traverse Internet in cleartext

SSL

- provides encrypted TCP connection
- data integrity
- end-point authentication

SSL is at app layer

- Apps use SSL libraries, which "talk" to TCP

SSL socket API

- cleartext passwds sent into socket traverse Internet encrypted
- See Chapter 8

Chapter 2: outline

2.2 Web and HTTP

Web and HTTP

First, a review…

- *web page* consists of *objects*
- object can be HTML file, JPEG image, Java applet, audio file,…
- web page consists of *base HTML-file* which includes *several referenced objects*
- each object is addressable by a *URL,* e.g.,

`www.someschool.edu/someDept/pic.gif`

host name | path name

HTTP overview

HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - *server:* Web server sends (using HTTP protocol) objects in response to requests

HTTP overview (continued)

uses TCP:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is "stateless"

- server maintains no information about past client requests

aside

protocols that maintain "state" are complex!

- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

HTTP connections

non-persistent HTTP

- at most one object sent over TCP connection
  - connection then closed
- downloading multiple objects required multiple connections

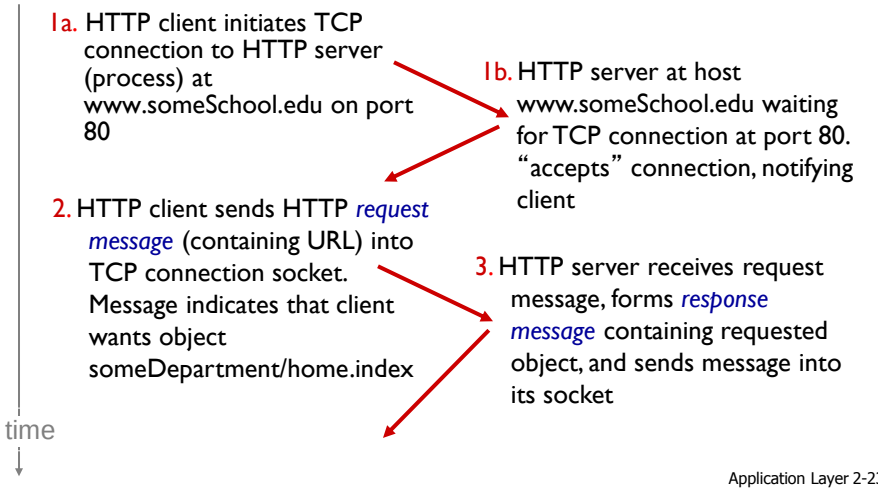
persistent HTTP

- multiple objects can be sent over single TCP connection between client, server

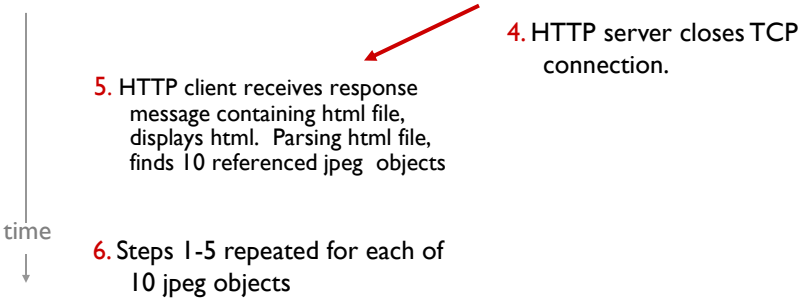
Non-persistent HTTP

suppose user enters URL:
`www.someSchool.edu/someDepartment/home.index`

(contains text, references to 10 jpeg images)



Non-persistent HTTP (cont.)



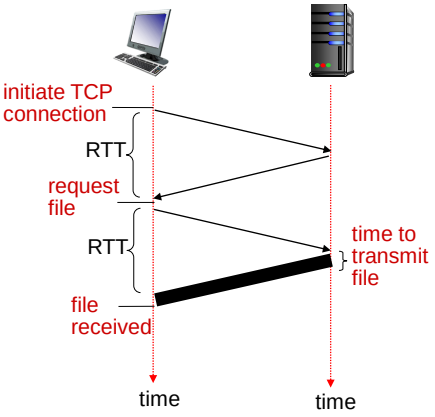
Non-persistent HTTP: response time

RTT (definition): time for a small packet to travel from client to server and back

HTTP response time:

- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- file transmission time
- non-persistent HTTP response time =

  2RTT+ file transmission time



Persistent HTTP

non-persistent HTTP issues:

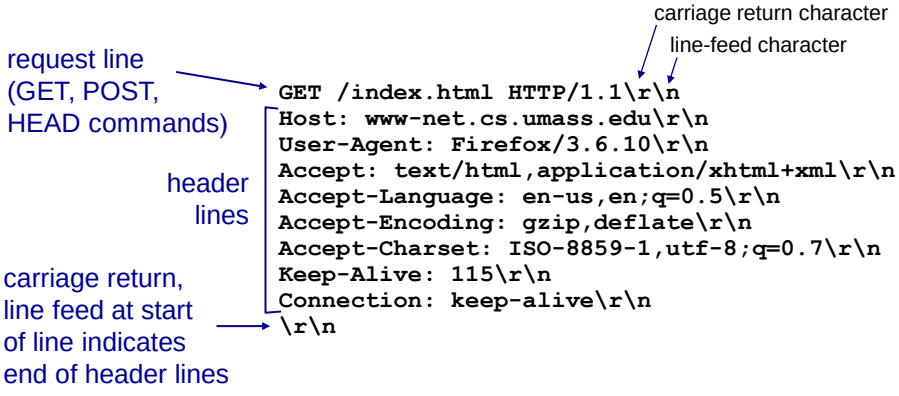
- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- browsers often open parallel TCP connections to fetch referenced objects

persistent HTTP:

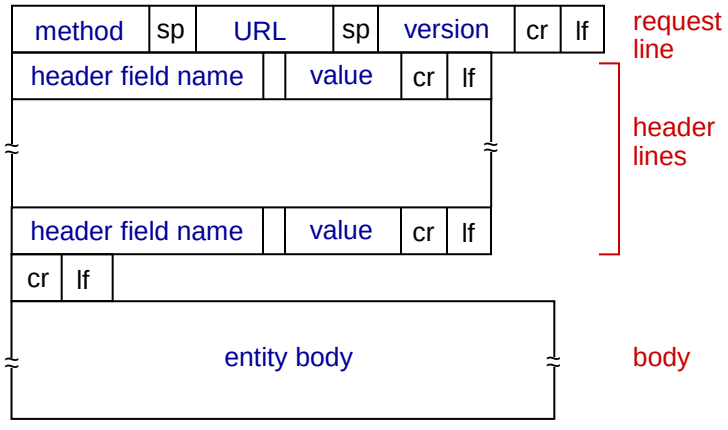
- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects

HTTP request message

- two types of HTTP messages: *request, response*
- HTTP request message:
  - ASCII (human-readable format)



HTTP request message: general format



Uploading form input

POST method:

- web page often includes form input
- input is uploaded to server in entity body

URL method:

- uses GET method
- input is uploaded in URL field of request line:

```
www.somesite.com/animalsearch?monkeys&banana
```


Method types

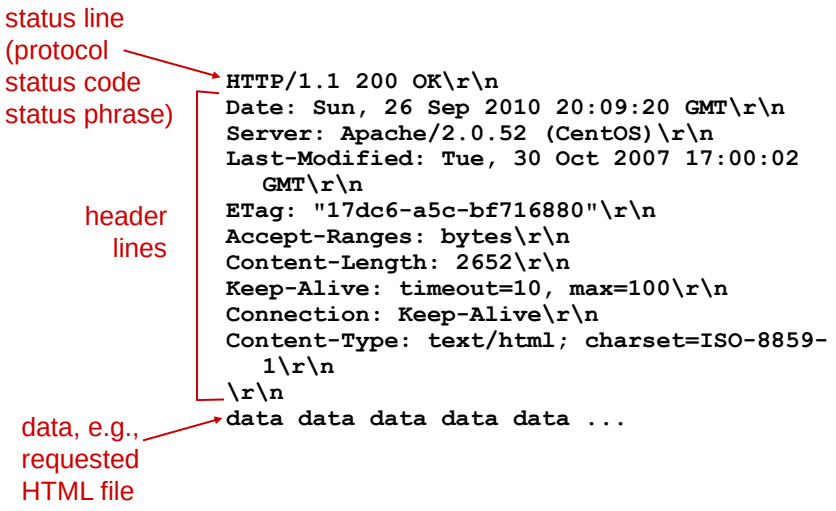
HTTP/1.0:

- GET
- POST
- HEAD
  - asks server to leave requested object out of response

HTTP/1.1:

- GET, POST, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field

HTTP response message



HTTP response status codes

- status code appears in 1st line in server-to-client response message.
- some sample codes:

200 OK
- request succeeded, requested object later in this msg

301 Moved Permanently
- requested object moved, new location specified later in this msg (Location:)

400 Bad Request
- request msg not understood by server

404 Not Found
- requested document not found on this server

505 HTTP Version Not Supported

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet cis.poly.edu 80
```

opens TCP connection to port 80 (default HTTP server port) at cis.poly.edu. anything typed in sent to port 80 at cis.poly.edu

2. type in a GET HTTP request:

```
GET /~ross/ HTTP/1.1
Host: cis.poly.edu
```

by typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server

3. look at response message sent by HTTP server!

User-server state: cookies

many Web sites use cookies

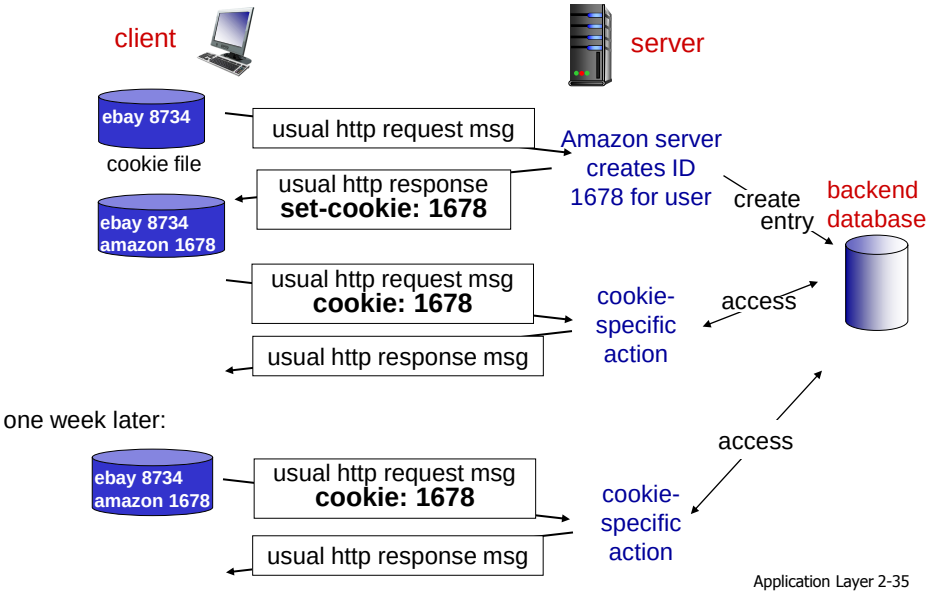
four components:

1) cookie header line of HTTP *response* message
2) cookie header line in next HTTP *request* message
3) cookie file kept on user's host, managed by user's browser
4) back-end database at Web site

example:

- Susan always access Internet from PC
- visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID
  - entry in backend database for ID

Cookies: keeping "state" (cont.)



Cookies (continued)

what cookies can be used for:

- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

aside

cookies and privacy:

- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites

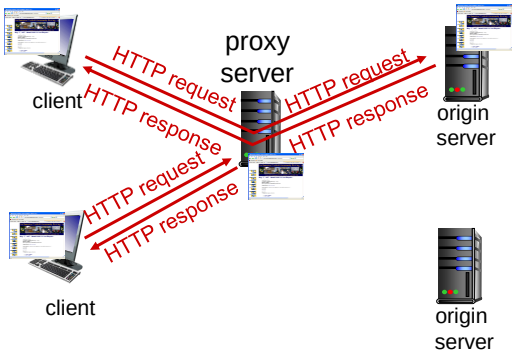
how to keep "state":

- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: http messages carry state

Web caches (proxy server)

goal: satisfy client request without involving origin server

- user sets browser: Web accesses via cache
- browser sends all HTTP requests to cache
  - object in cache: cache returns object
  - else cache requests object from origin server, then returns object to client



More about Web caching

- cache acts as both client and server
  - server for original requesting client
  - client to origin server
- typically cache is installed by ISP (university, company, residential ISP)

why Web caching?

- reduce response time for client request
- reduce traffic on an institution's access link
- Internet dense with caches: enables "poor" content providers to effectively deliver content (so too does P2P file sharing)

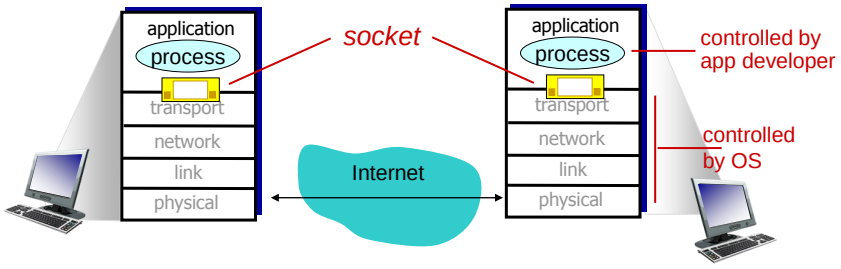
Chapter 2: outline

2.7 socket programming with UDP and TCP

Socket programming

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Socket programming

Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

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:

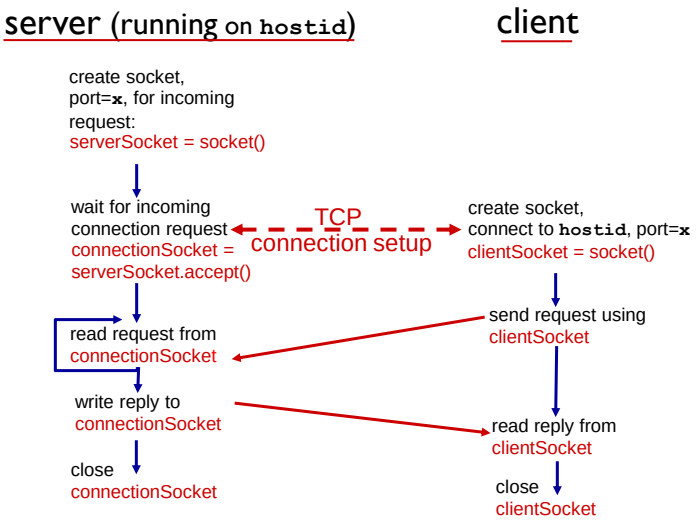
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- 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 (more in Chap 3)

application viewpoint:

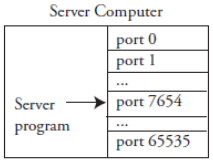
TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

Client/server socket interaction: TCP

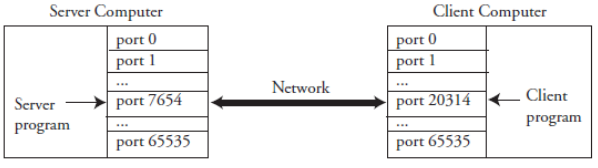


Client/Socket Interaction

1. The server listens and waits for a connection on port 7654.



2. The client connects to the server on port 7654. It uses a local port that is assigned automatically, in this case, port 20314.



The server program can now communicate over a socket bound locally to port 7654 and remotely to the client's address at port 20314.

The client program can now communicate over a socket bound locally to port 20314 and remotely to the server's address at port 7654.

TCP Server Example

```
import java.util.Date;
import java.net.ServerSocket;
import java.net.Socket;
import java.io.DataOutputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class DateServer
{
        public static void main(String[] args)
        {
                Date now = new Date();

                try
                {
                        System.out.println("Waiting for a connection on port 7654.");
                        ServerSocket serverSock = new ServerSocket(7654);
                        Socket connectionSock = serverSock.accept();

                        BufferedReader clientInput = new BufferedReader(
                                new InputStreamReader(connectionSock.getInputStream()));
                        DataOutputStream clientOutput = new DataOutputStream(
                                connectionSock.getOutputStream());
```

TCP Server Example

```
                        System.out.println("Connection made, waiting for client to send their name.");
                        String clientText = clientInput.readLine();
                        String replyText = "Welcome, " + clientText +
                                ", Today is " + now.toString() + "\n";
                        clientOutput.writeBytes(replyText);
                        System.out.println("Sent: " + replyText);

                        clientOutput.close();
                        clientInput.close();
                        connectionSock.close();
                        serverSock.close();
                }
                catch (IOException e)
                {
                        System.out.println(e.getMessage());
                }
        }
}
```

TCP Client Example

```
import java.net.Socket;
import java.io.DataOutputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class DateClient
{
        public static void main(String[] args)
        {
                try
                {
                        String hostname = "localhost";
                        int port = 7654;

                        System.out.println("Connecting to server on port " + port);
                        Socket connectionSock = new Socket(hostname, port);

                        BufferedReader serverInput = new BufferedReader(
                                new InputStreamReader(connectionSock.getInputStream()));
                        DataOutputStream serverOutput = new DataOutputStream(
                                connectionSock.getOutputStream());
```


TCP Client Example

```
                        System.out.println("Connection made, sending name.");
                        serverOutput.writeBytes("Dusty Rhodes\n");

                        System.out.println("Waiting for reply.");
                        String serverData = serverInput.readLine();
                        System.out.println("Received: " + serverData);

                        serverOutput.close();
                        serverInput.close();
                        connectionSock.close();
                }
                catch (IOException e)
                {
                        System.out.println(e.getMessage());
                }
        }
}
```

Problem: Blocking Calls

- Only good for one invocation
- The server process blocks at the accept() call

- Solution: Threading
- To make a thread in Java:
  - Make your class extend "Thread"
  - Make a public void run() method that does the work of the thread
  - Invoke the thread by invoking the start() method of the instance of the class

Simple Thread

```
public class SimpleThread extends Thread
{
        private int id;

        public SimpleThread(int id)
        {
                this.id = id;
        }

        public void run()
        {
                System.out.println("Hello from thread " + id);
        }

        public static void main(String[] args)
        {
                SimpleThread thread1 = new SimpleThread(1);
                SimpleThread thread2 = new SimpleThread(2);

                thread1.start();
                thread2.start();
        }

}
```

Threaded DateServer

```
import java.util.Date;
import java.net.ServerSocket;
import java.net.Socket;
import java.io.DataOutputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class ThreadedDateServer extends Thread
{
        private Socket connectionSock = null;

        public ThreadedDateServer(Socket theSock)
        {
                connectionSock = theSock;
        }
```


Threaded DateServer

```
public void run()
        {
                try
                {
                        Date now = new Date();

                        BufferedReader clientInput = new BufferedReader(
                                new InputStreamReader(connectionSock.getInputStream()));
                        DataOutputStream clientOutput = new DataOutputStream(
                                connectionSock.getOutputStream());

                        System.out.println("Connection made, waiting for client to send their name.");
                        String clientText = clientInput.readLine();
                        String replyText = "Welcome, " + clientText + ", Today is " + now.toString() + "\n";
                        clientOutput.writeBytes(replyText);
                        System.out.println("Sent: " + replyText);

                        clientOutput.close();
                        clientInput.close();
                        connectionSock.close();
                }
                catch (IOException e)
                {
                        System.out.println(e.getMessage());
                }
        }
```

Threaded DateServer

```
        public static void main(String[] args)
        {
                try
                {
                        System.out.println("Waiting for a connection on port 7654.");
                        ServerSocket serverSock = new ServerSocket(7654);
                        while (true)
                        {
                          Socket connectionSock = serverSock.accept();

                          // Make thread
                          ThreadedDateServer dateThread = new ThreadedDateServer(connectionSock);
                          dateThread.start();
                        }
                        //serverSock.close();          // Never gets here in this simple version
                }
                catch (IOException e)
                {
                        System.out.println(e.getMessage());
                }
        }
}
```