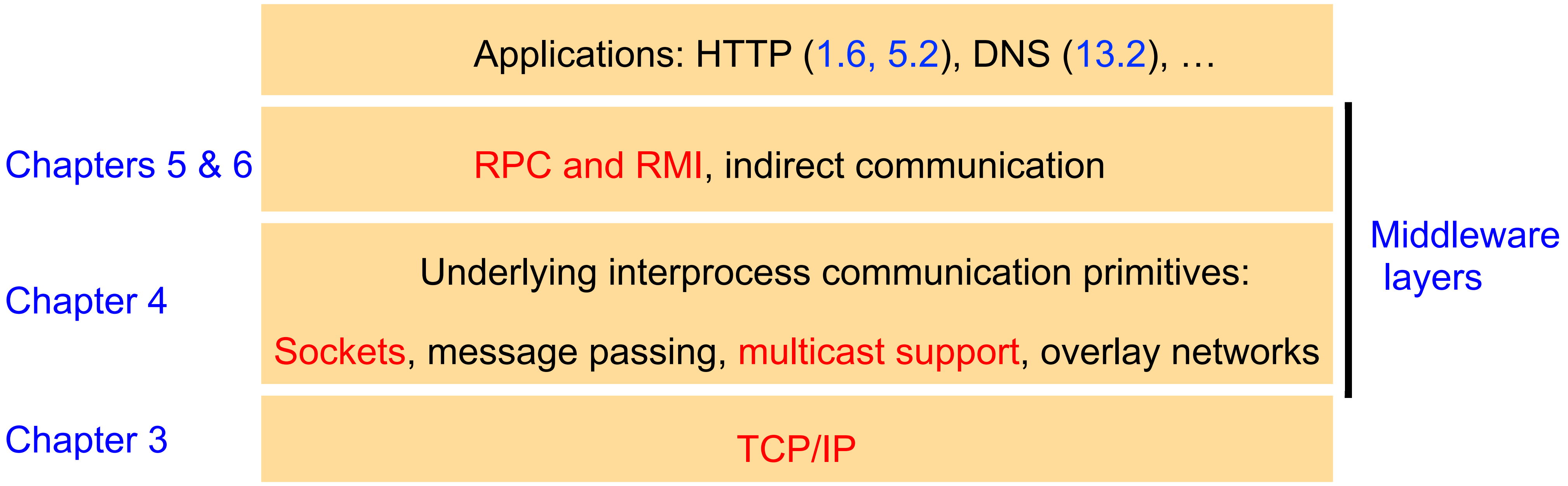
1

Interprocess Communication

CMPS 4760/6760: Distributed Systems

Acknowledgement: slides adapted from the slides accompanied by the book: James Kurose and Keith Ross, Computer Networking: A Top-Down Approach (7th edition), Pearson, 2016

Outline

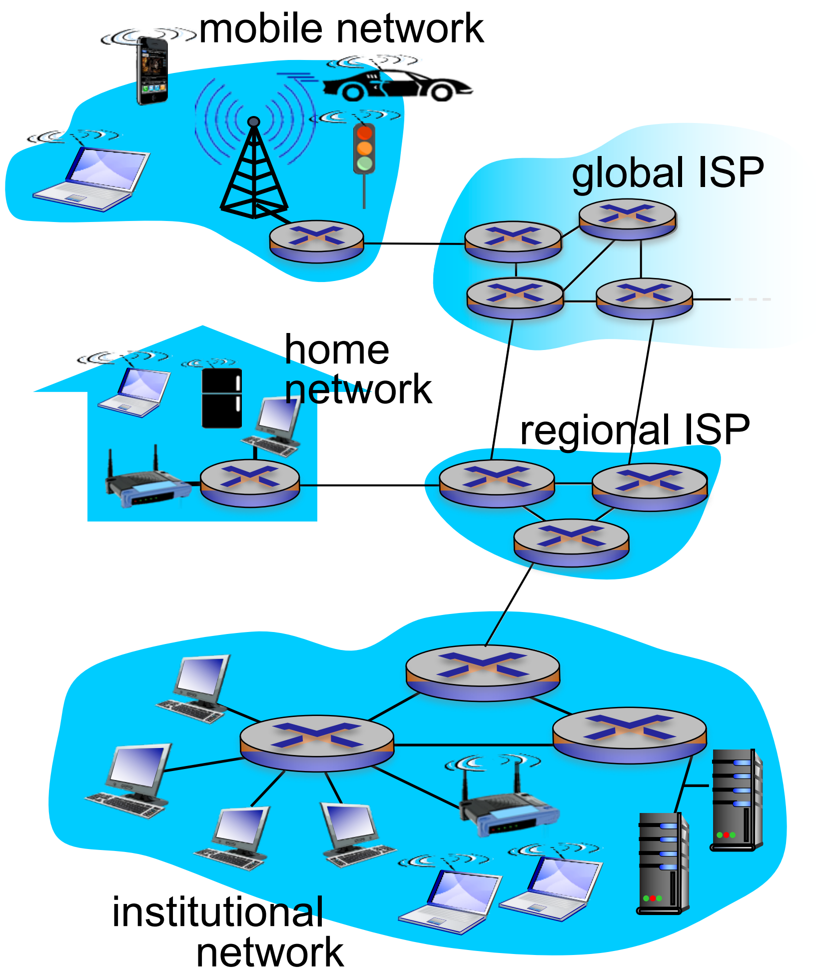


Applications: HTTP (1.6, 5.2), DNS (13.2), …
Chapters 5 & 6
RPC and RMI, indirect communication
Chapter 4
Underlying interprocess communication primitives:
Sockets, message passing, multicast support, overlay networks
Middleware layers
Chapter 3
TCP/IP

An Overview of the Internet

- Packet switching
- Performance
- Internet protocol stack
- Network layer: IP
- Transport layer: UDP, TCP
- Application layer: HTTP, DNS

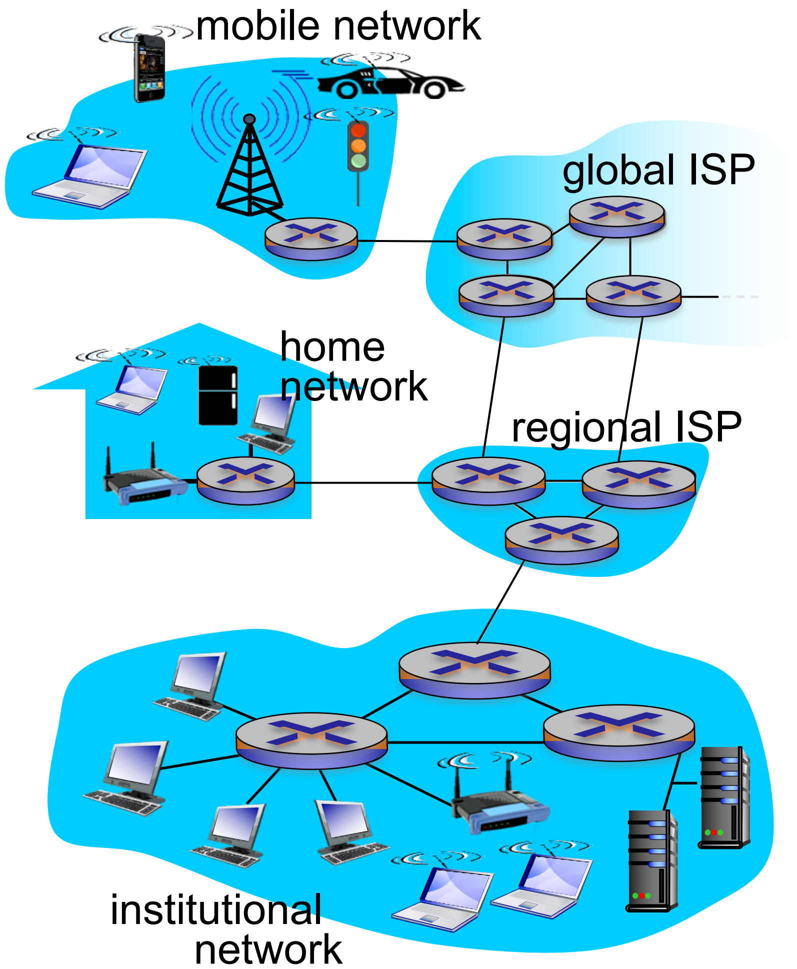
A Nuts-and-Bolts View of the Internet



- Hosts = end systems
  - Running network apps
  - Billions of connected computing devices
- Communication links
  - copper, cables, fiber, radio, satellite
  - transmission rate (bit/sec), maximum distance
- Packet switches: forward packets
  - Routers and link-layer switches
  - ISP: a network of packet switches
- Internet: “network of networks”

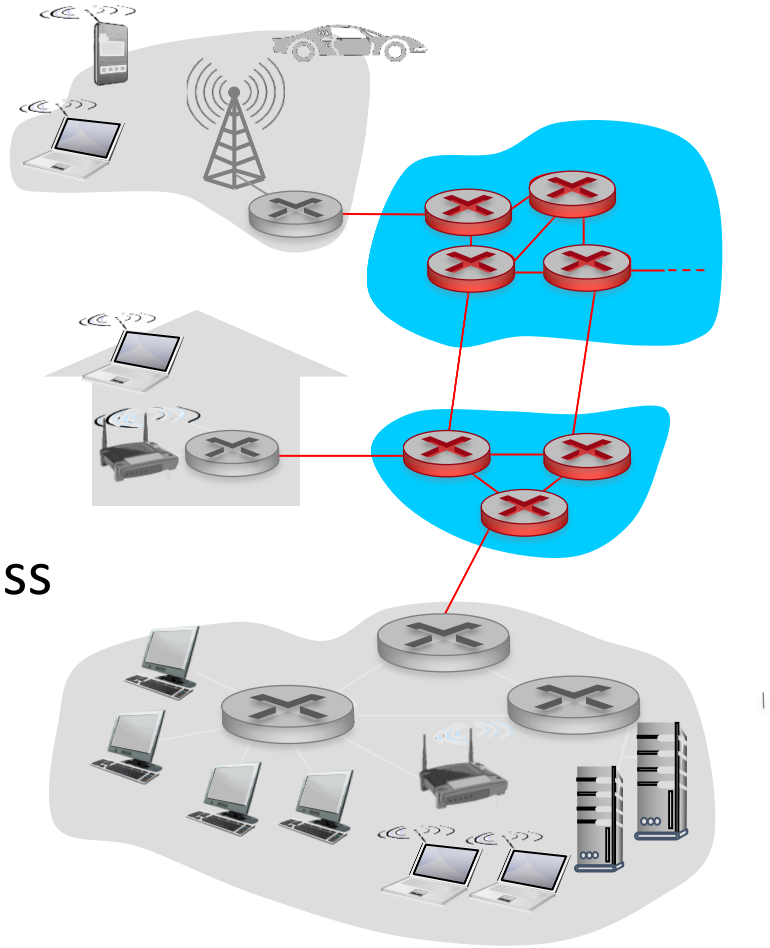
A closer look at network structure

- Network Core
  - Interconnected routers
- Network Edge
  - access networks: connect hosts to the core
    - DSL, Cable, Ethernet, Wireless, Fiber to the home (FTTH), Satellite
  - hosts: clients and servers
    - clients: desktops, smartphones, smart devices
    - servers: service/content providers, often in data centers



The Network Core

- mesh of interconnected routers
- packet-switching: hosts break application-layer messages into *packets*
  - A packet: header + payload (a set of bits)
  - forward packets from one router to the next, across links on path from source to destination
  - each packet transmitted at full link capacity

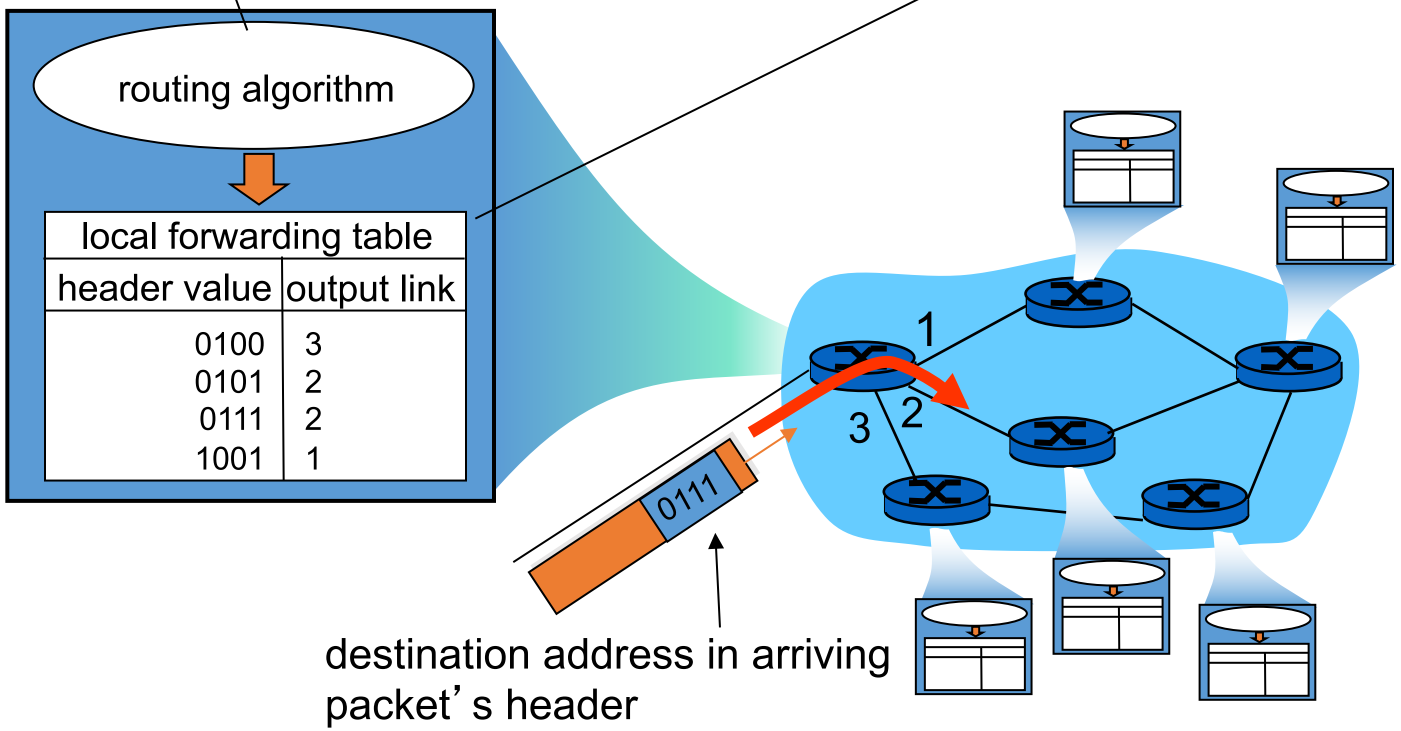


Key network-core functions

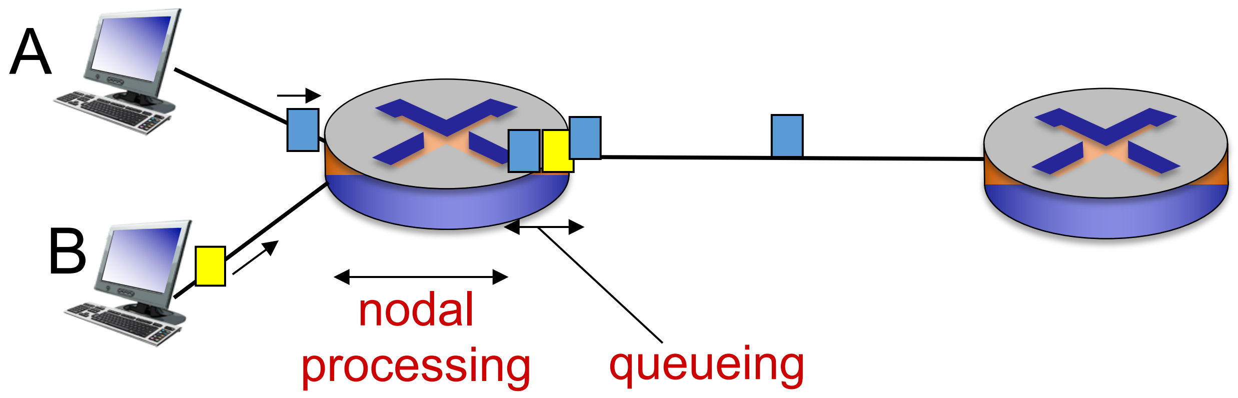
routing: determines source-destination route taken by packets

- *routing algorithms*

forwarding: move packets from router's input to appropriate router output



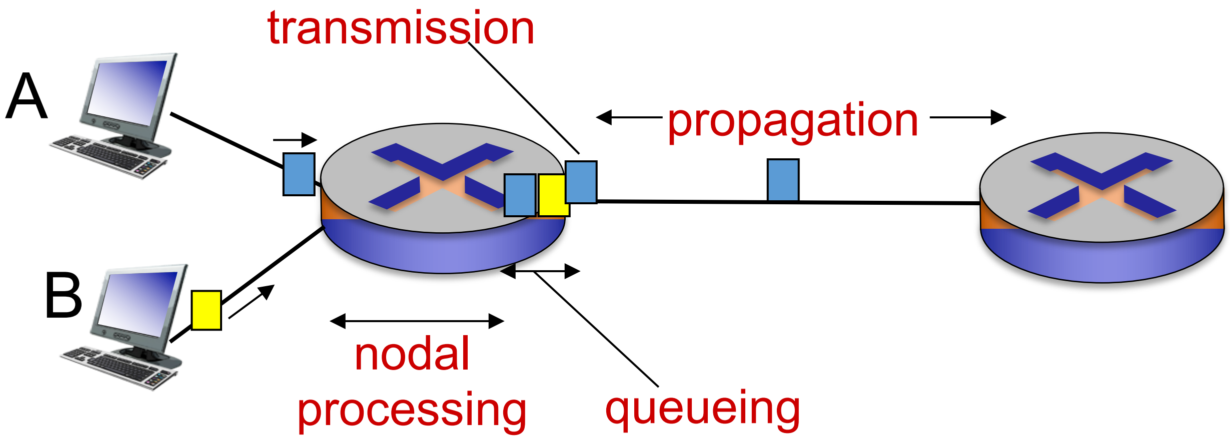
Four sources of packet delay



d_{proc}: nodal processing
- check bit errors
- determine output link
- typically < msec

d_{queue}: queueing delay
- time waiting at output link for transmission
- depends on congestion level of router

Four sources of packet delay



$$d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop}$$

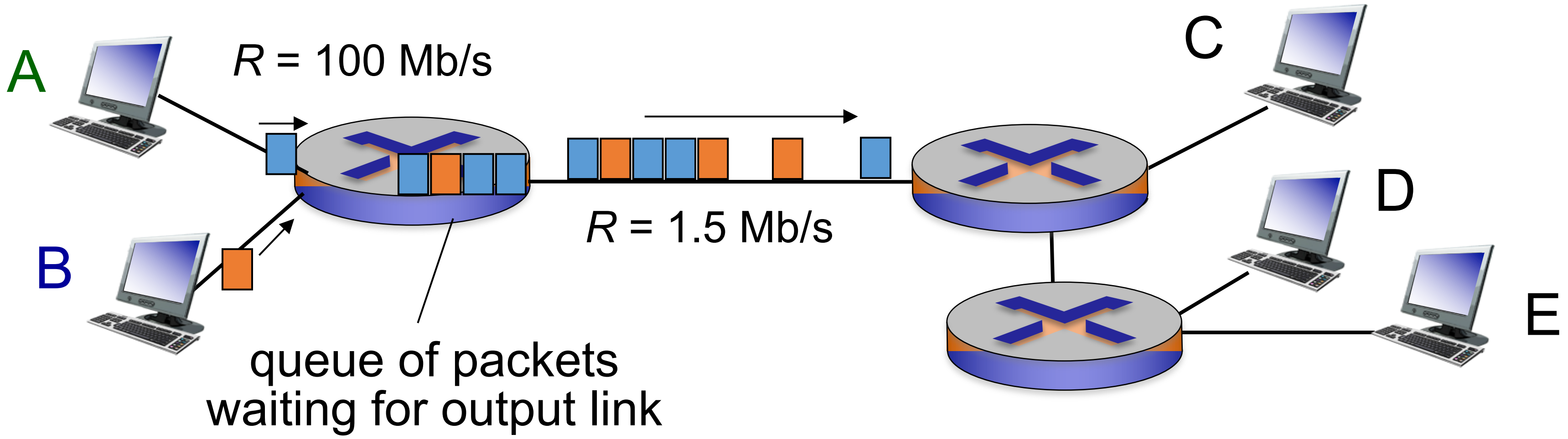
d_{trans}: transmission delay:

- L: packet length (bits)
- R: link *bandwidth (bps)*
- $d_{trans} = L/R$

d_{prop}: propagation delay:

- d: length of physical link
- s: propagation speed (~2×10^8 m/sec)
- $d_{prop} = d/s$

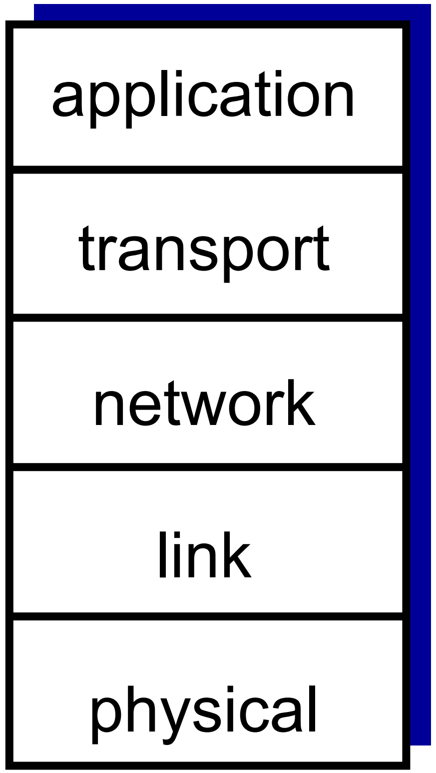
Queueing and packet loss



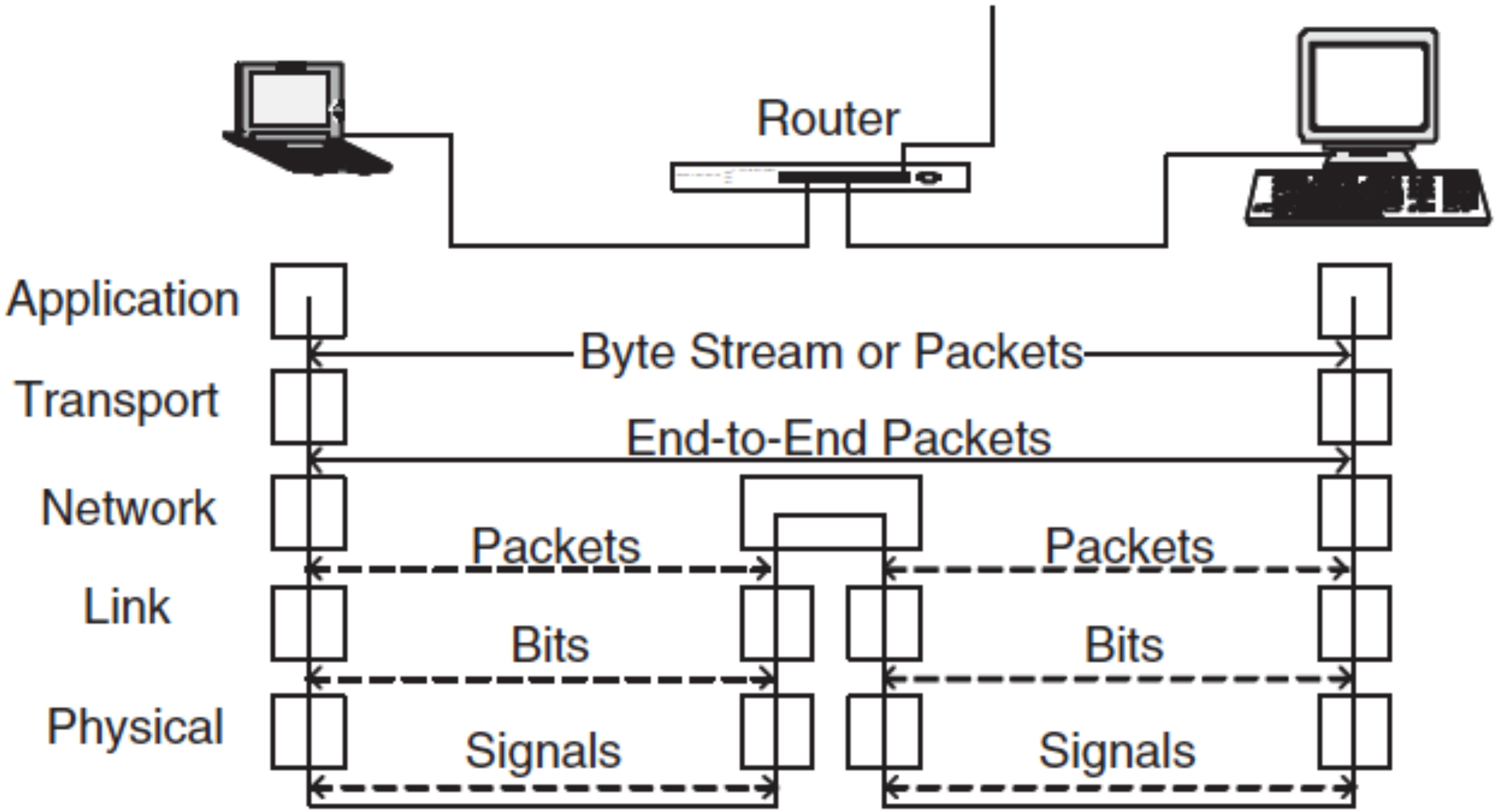
- Each output link has a queue (buffer) of finite space
- An arriving packet will queue when link is busy
- Packet loss will occur when the output queue is full

Internet protocol stack

- *application:* supporting network applications
  - HTTP, SMTP, FTP,…
- *transport:* process-process data transfer
  - TCP, UDP
- *network:* routing of datagrams from source to destination
  - IP
- *link:* data transfer between neighboring network elements
  - Ethernet, WiFi, …
- *physical:* bits “on the wire”

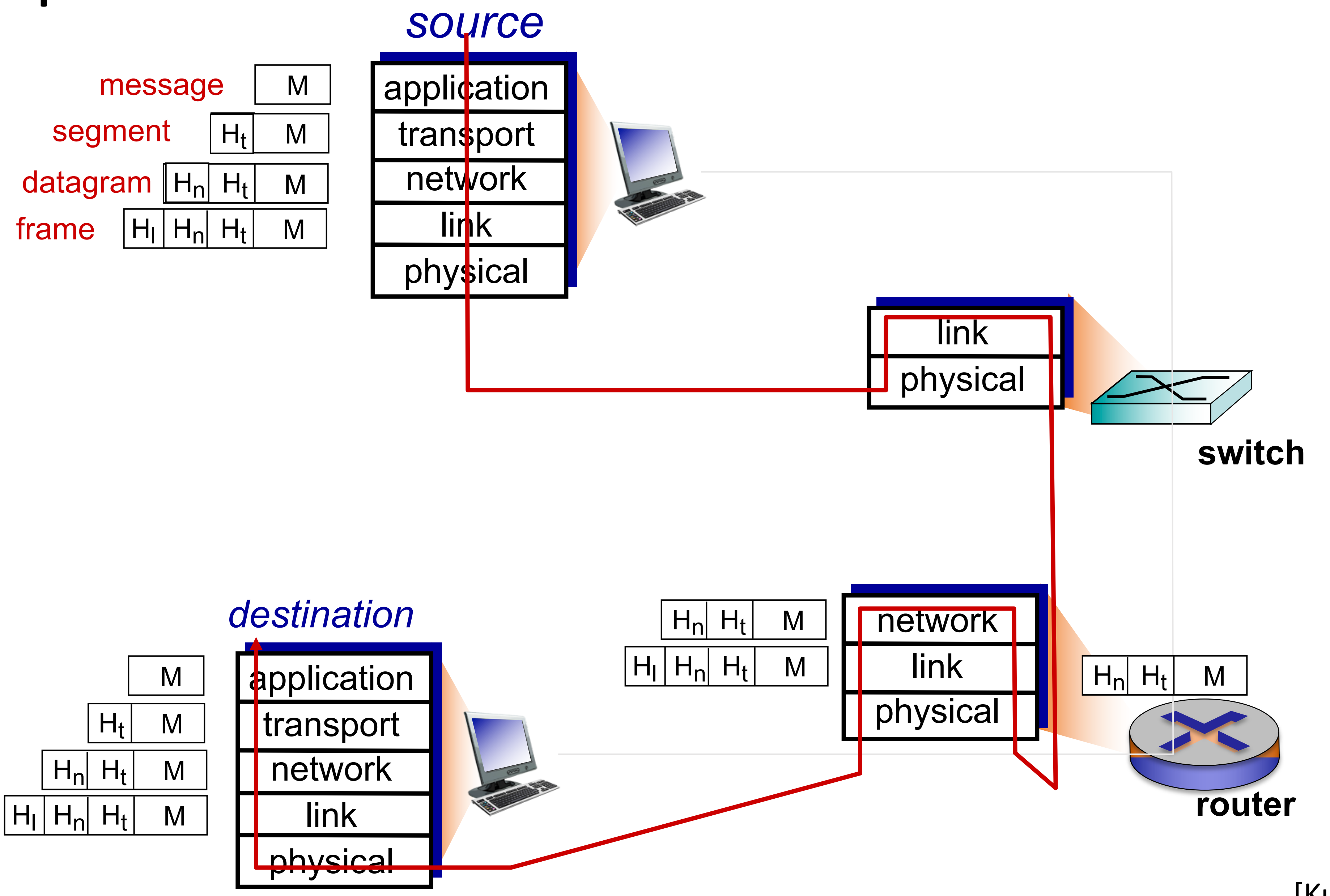


Internet protocol stack



[Walrand and Parekh]

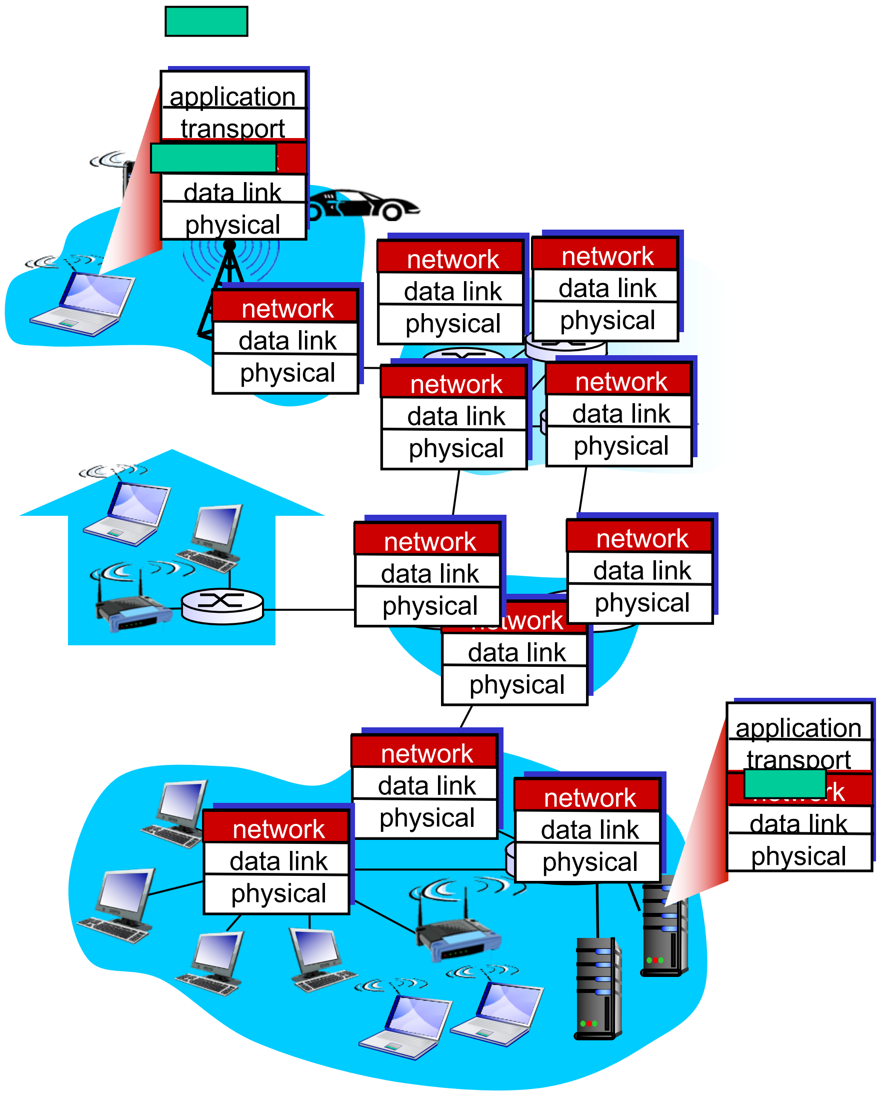
Encapsulation



[Kurose and Ross]

Network Layer

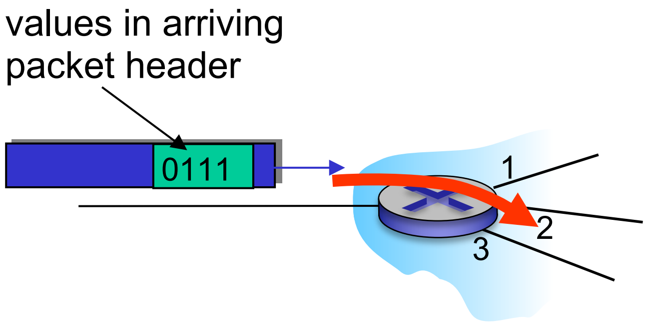
- transport segment from sending to receiving host
- network layer protocols in *every* host & router
- router examines header fields in all IP datagrams passing through it
- *The Internet's network layer provides "best-effort" service*



Network layer: data plane, control plane

Data plane

- local, per-router function
  - forwarding
  - dropping
  - modify field
  - …

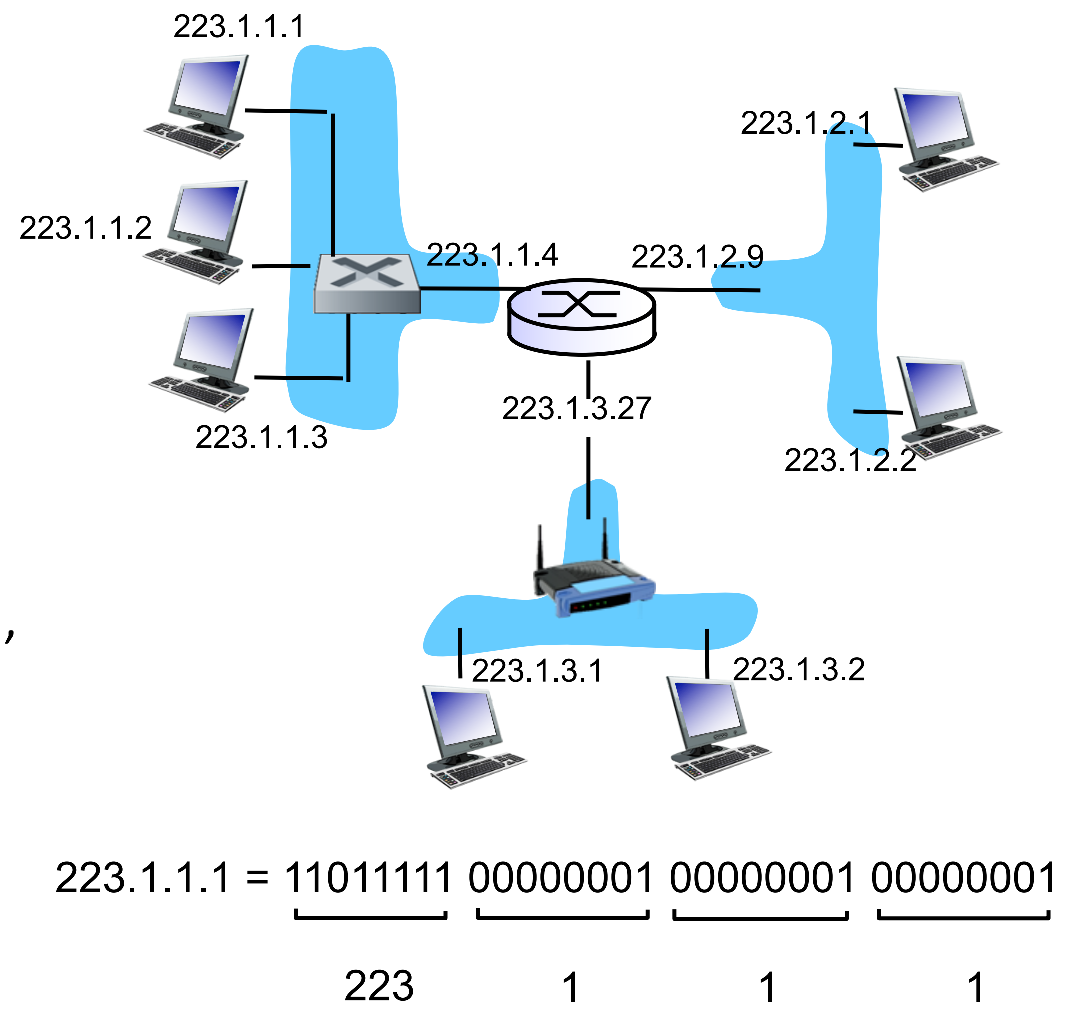


Control plane

- network-wide logic
  - routing
  - access control
  - load balancing
  - …
- two control-plane approaches:
  - *traditional routing algorithms:* implemented in routers
  - *software-defined networking (SDN)*: implemented in (remote) servers

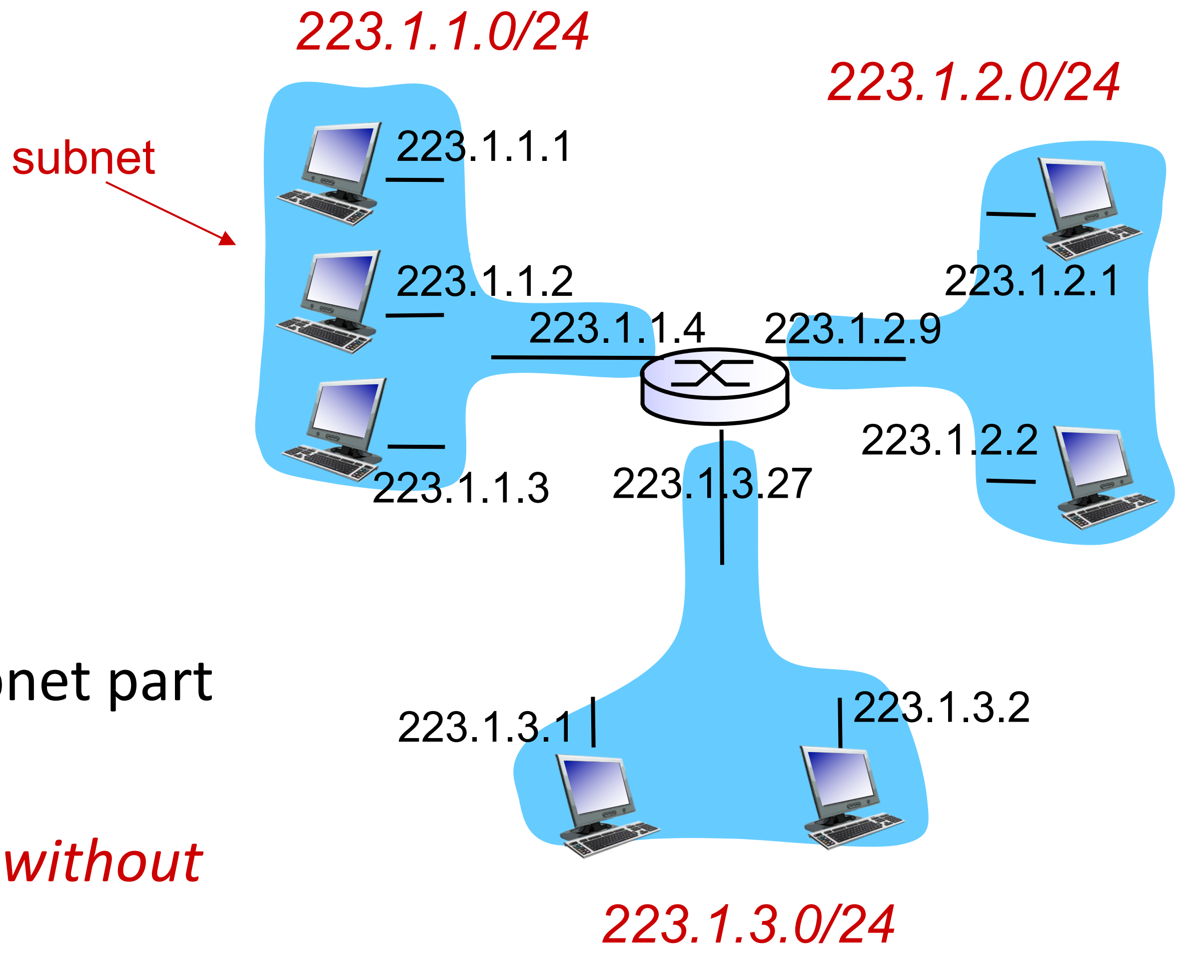
IP addressing: introduction

- *IP address:* 32-bit identifier for host, router *interface*
- *interface:* boundary between host/router and physical link
  - routers typically have multiple interfaces
  - host typically has one or two interfaces (e.g., wired Ethernet, wireless 802.11)
- *IP addresses associated with each interface*



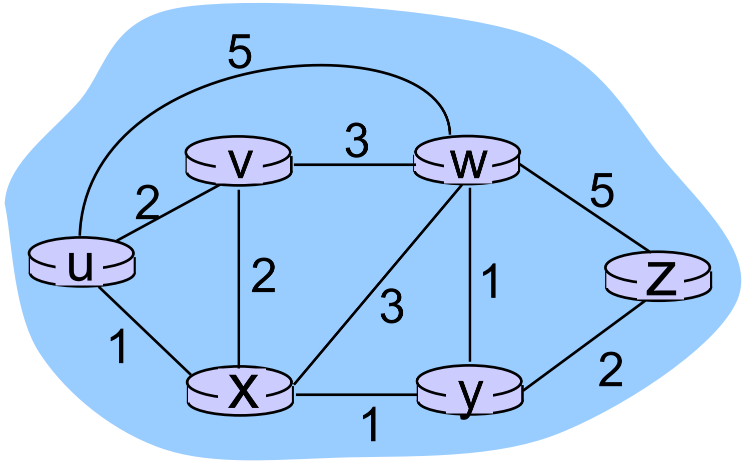
Subnets

- IP address:
  - subnet part - high order bits
  - host part - low order bits
- *what's a subnet ?*
  - device interfaces with same subnet part of IP address
  - can physically reach each other *without intervening router*



subnet mask: /24

Routing: graph abstraction



c(x,x') = cost of link (x,x')

e.g., c(w,z) = 5

cost could always be 1, or inversely related to bandwidth, or related to congestion or delay

cost of path $(x_1, x_2, x_3, \ldots, x_p) = c(x_1,x_2) + c(x_2,x_3) + \ldots + c(x_{p-1},x_p)$

key question: what is the least-cost path between u and z ?

routing algorithm: algorithm that finds that least cost path

Making routing scalable

our routing study thus far - idealized

- all routers identical
- network "flat"

… *not* true in practice

scale: with billions of destinations:

- can't store all destinations in routing tables!
- routing table exchange would swamp links!

administrative autonomy

- internet = network of networks
- each network admin may want to control routing in its own network

Internet approach to scalable routing

aggregate routers into regions known as "autonomous systems" (AS) (a.k.a. "domains")

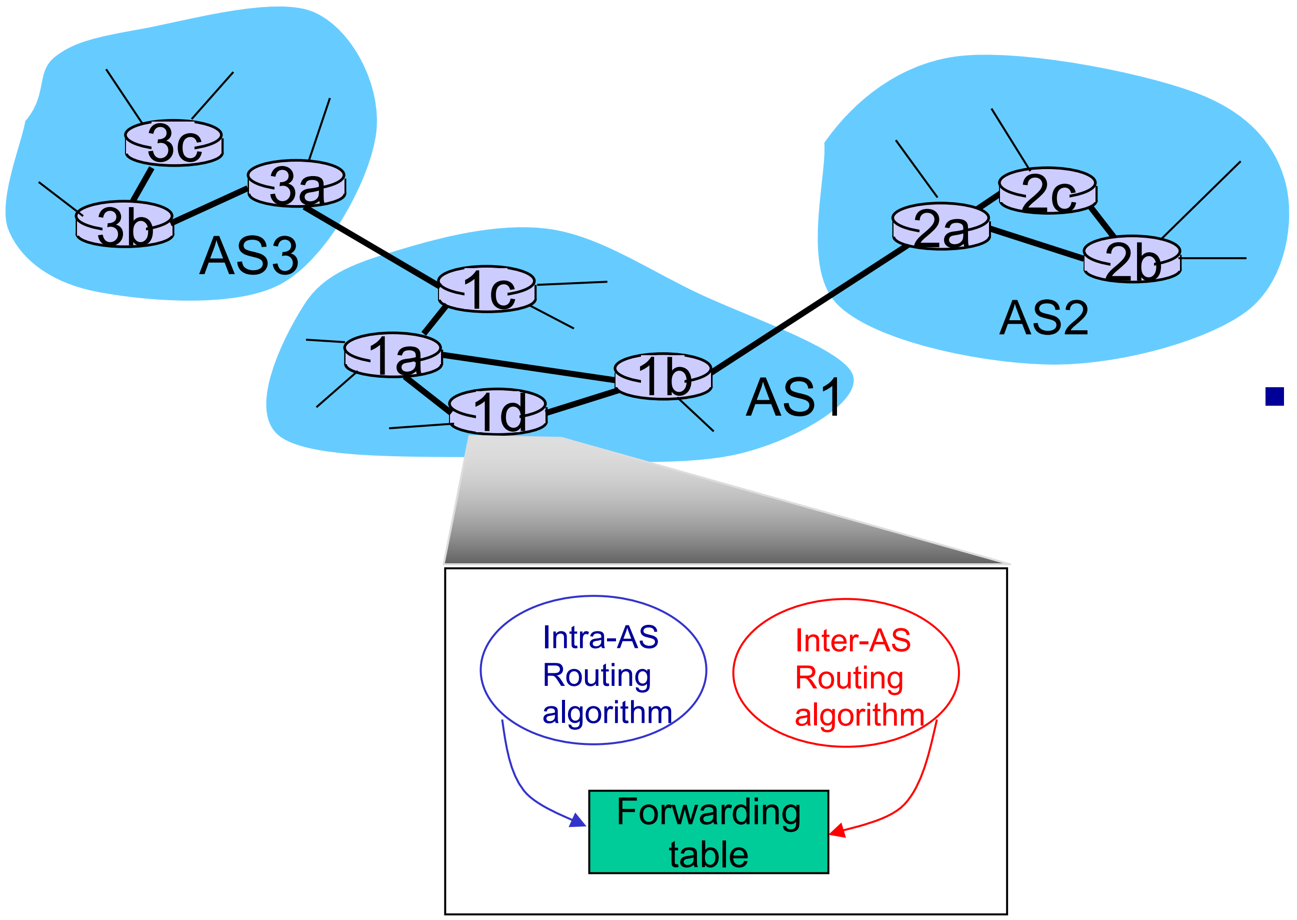
intra-AS routing

- routing among hosts, routers in same AS ("network")
- all routers in AS must run *same* intra-domain protocol
- routers in *different* AS can run *different* intra-domain routing protocol

inter-AS routing

- routing among AS'es
- gateway router: at "edge" of its own AS, has link(s) to router(s) in other AS'es
- gateways perform inter-domain routing (as well as intra-domain routing)

Interconnected ASes



- forwarding table configured by both intra- and inter-AS routing algorithm
  - intra-AS routing determine entries for destinations within AS
  - inter-AS & intra-AS determine entries for external destinations

NAT: network address translation

- IPv4 has ~4.3 billion IP addresses, but we have
  - ~7.6 billion people in 2018, each with multiple devices
  - ~30 billion Internet of Things (IoT) devices in 2020

- *motivation:* local network uses just one IP address as far as outside world is concerned:
  - range of addresses not needed from ISP: just one IP address for all devices
  - can change addresses of devices in local network without notifying outside world
  - devices inside local net not explicitly addressable, visible by outside world (a security plus)

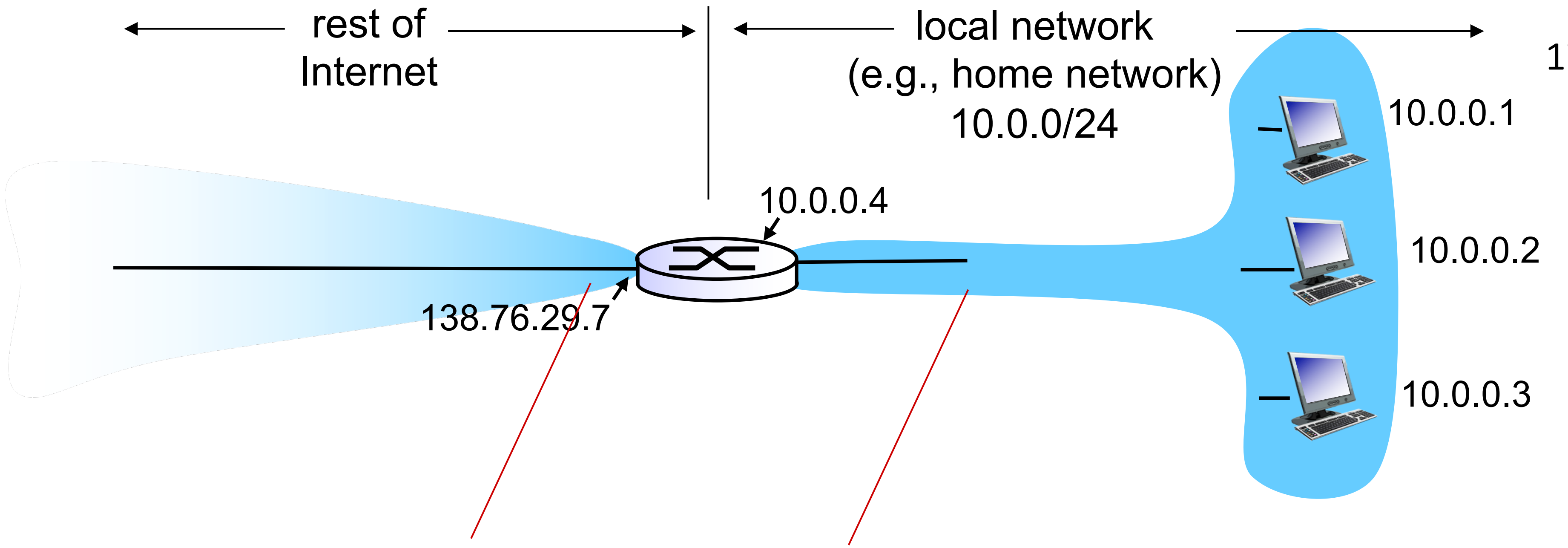
NAT: network address translation

Private IP addresses:

10.x.x.x

192.168.x.x

172.16.0.0 – 172.31.255.255



all datagrams *leaving* local network have *same* single source NAT IP address: 138.76.29.7,different source port numbers

datagrams with source or destination in this network have 10.0.0/24 address for source, destination (as usual)

IPv6

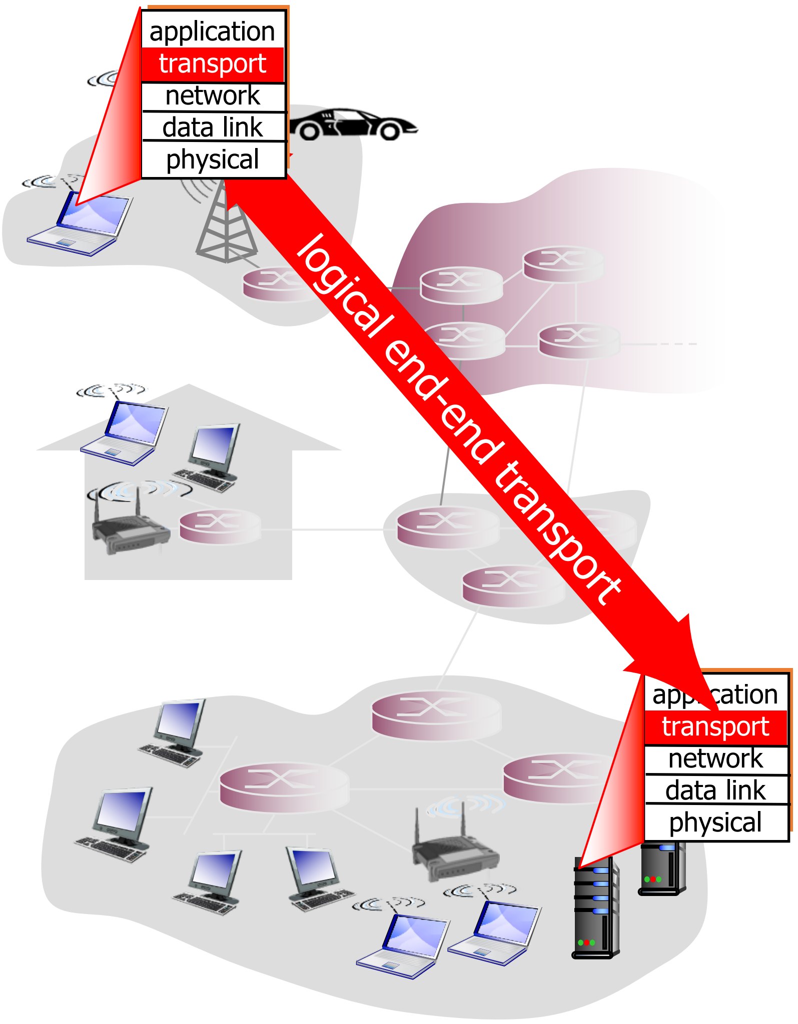
- *initial motivation:* 32-bit address space soon to be completely allocated.
- additional motivation:
  - header format helps speed processing/forwarding
  - header changes to facilitate QoS

IPv6 datagram format:

- 128-bit address space
- fixed-length 40 byte header
- no fragmentation allowed

Transport layer

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP

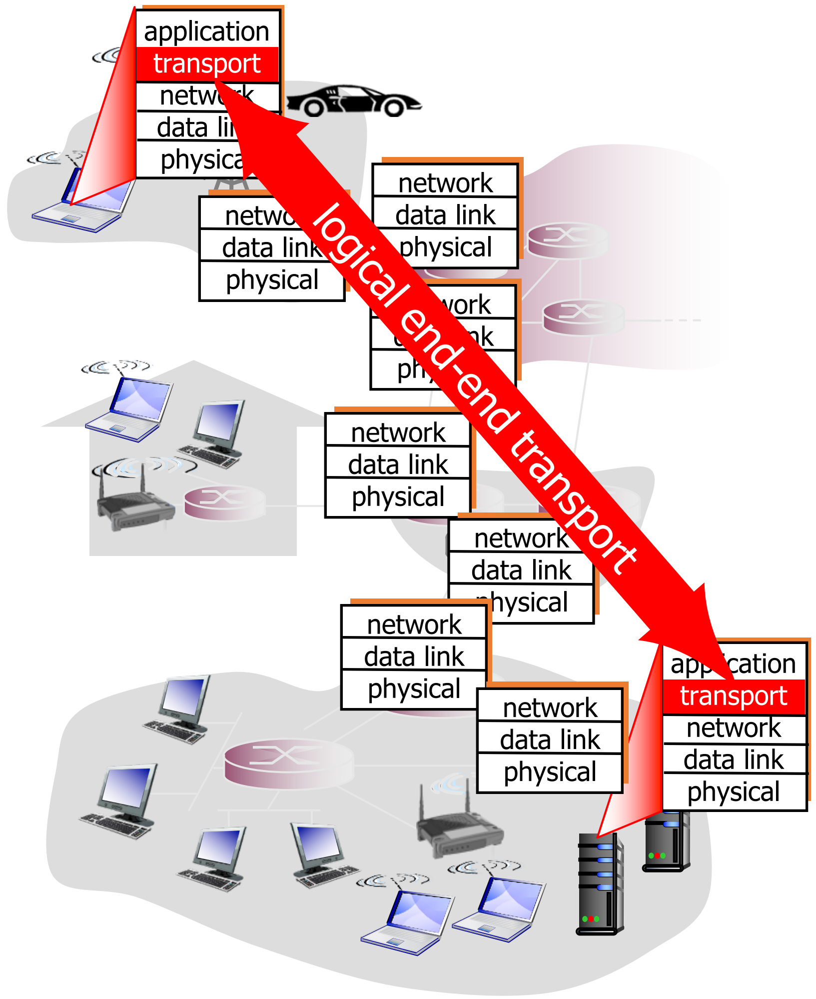


Transport vs. network layer

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

Internet transport-layer protocols

- unreliable, unordered delivery: UDP
  - connectionless
  - no-frills extension of "best-effort" IP
- reliable, in-order delivery (TCP)
  - connection-oriented: 3-way handshake
  - flow control
  - congestion control
- services not available:
  - delay guarantees
  - bandwidth guarantees



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

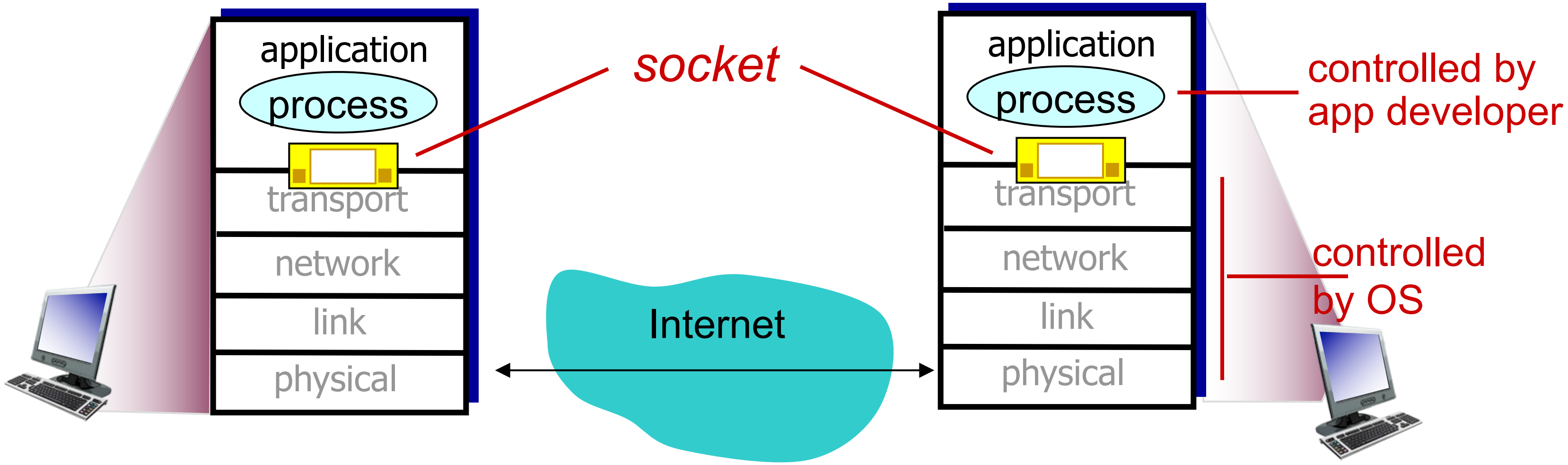
Addressing processes

- 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

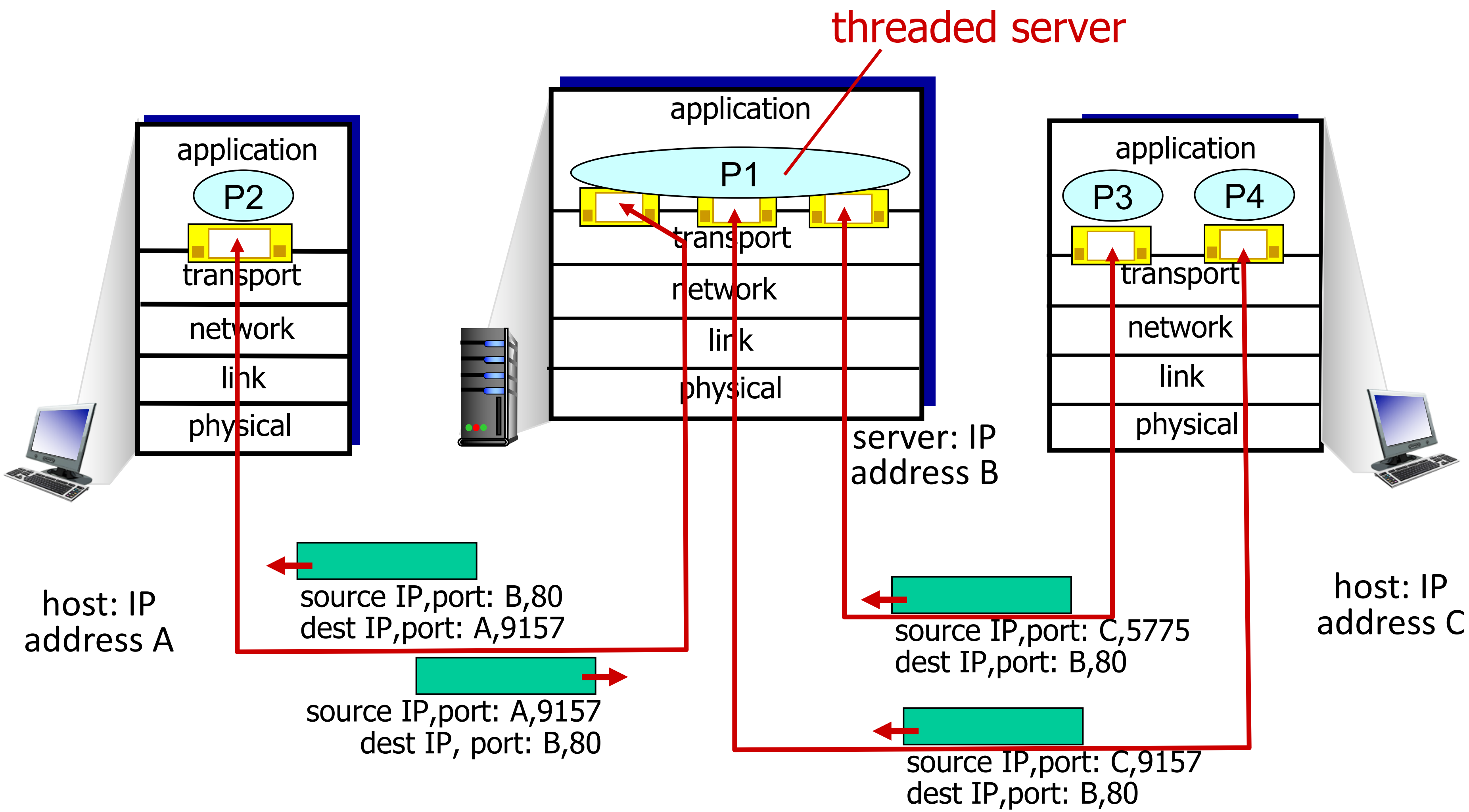
- *identifier* includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to cs.tulane.edu web server:
  - IP address: 129.81.226.25
  - port number: 80

Socket

- 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

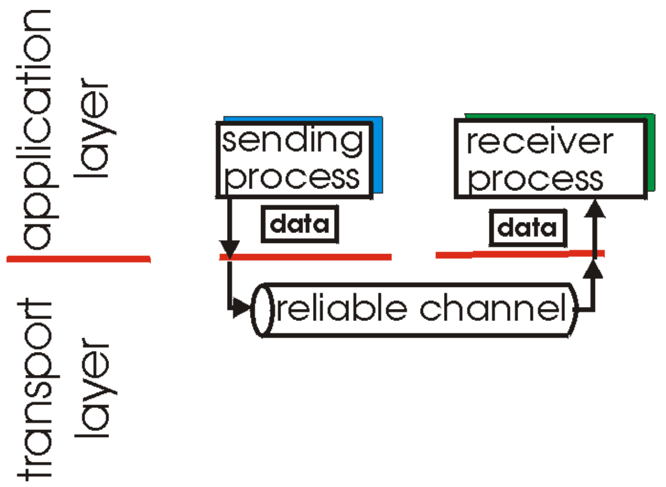


TCP multiplexing and demultiplexing



Principles of reliable data transfer

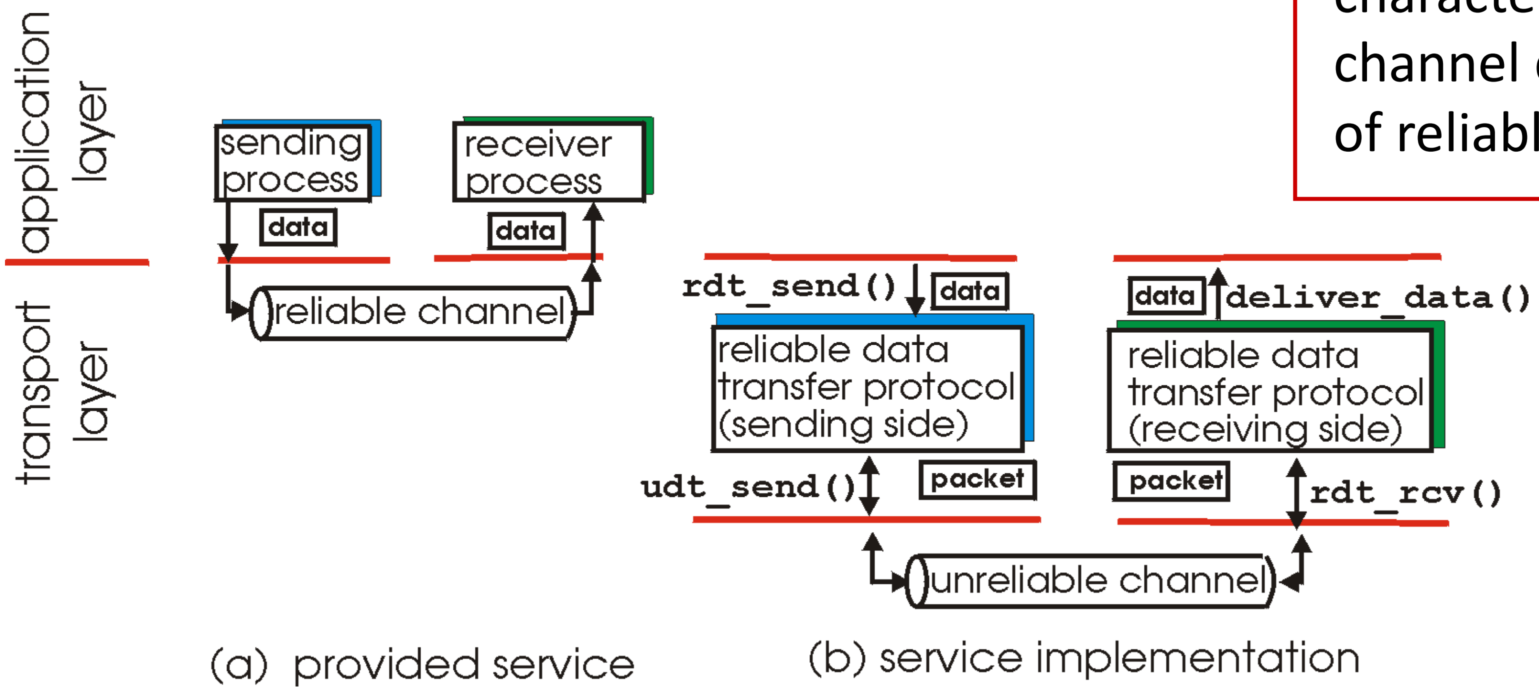
- important in application, transport, link layers
  - top-10 list of important networking topics!



(a) provided service

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!



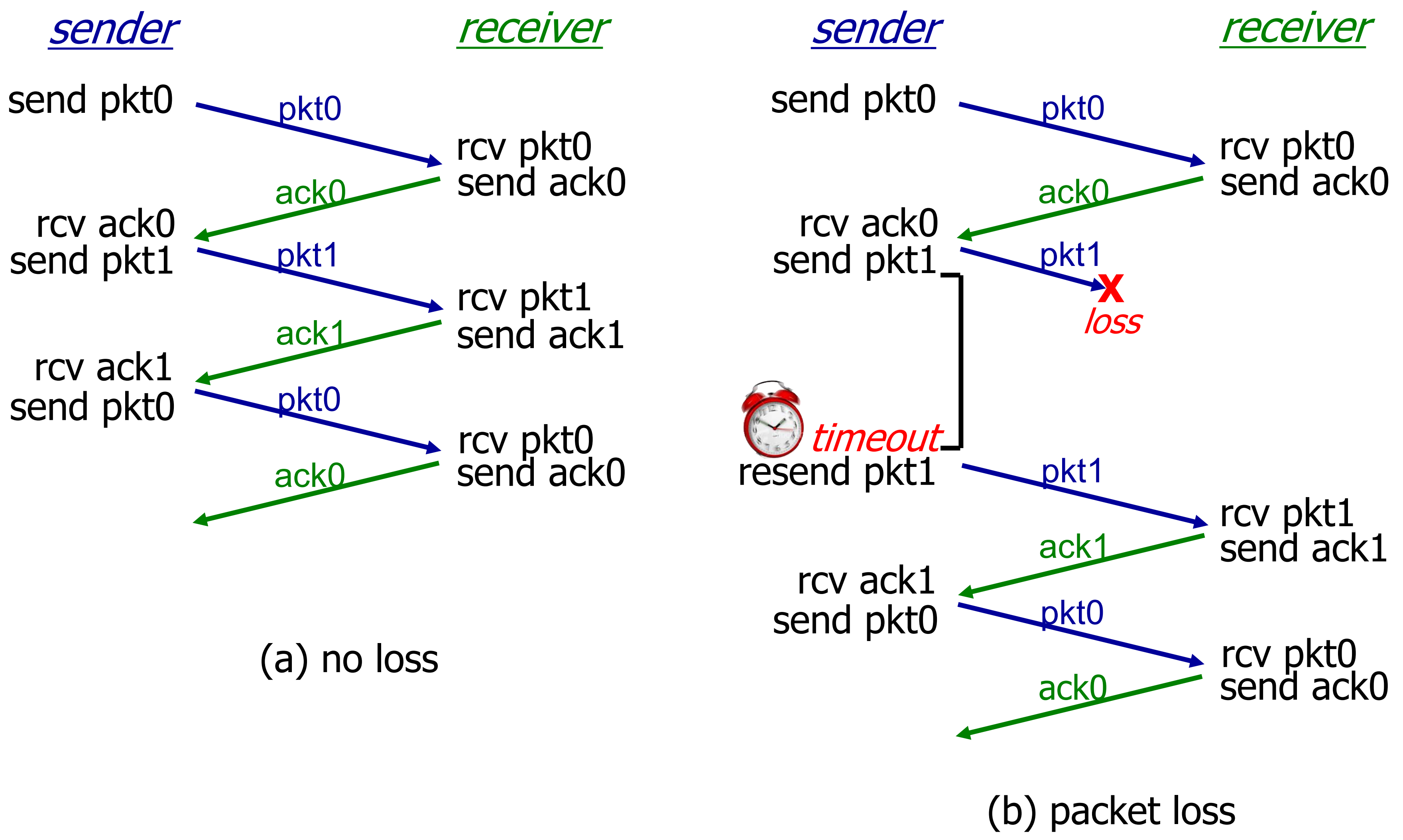
characteristics of unreliable channel determine complexity of reliable data transfer protocol

Potential Channel Errors

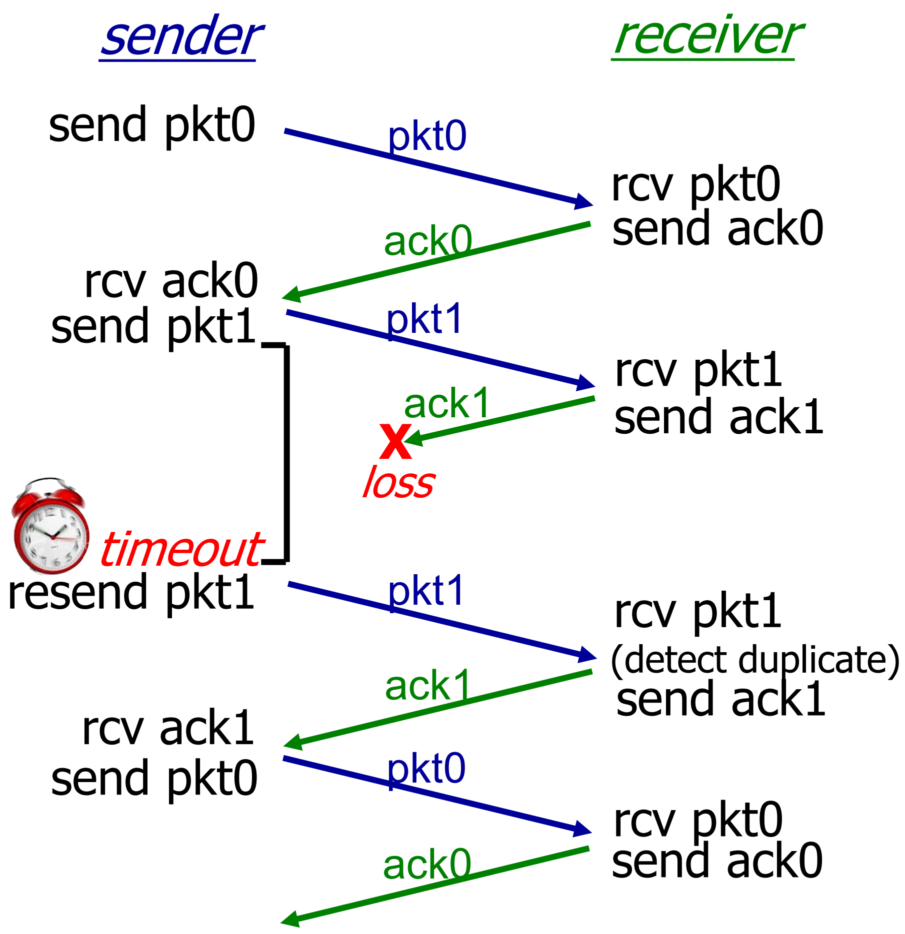
- bit errors
- loss (drop) of packets
- reordering or duplication

➢ characteristics of unreliable channel determine complexity of reliable data transfer protocol

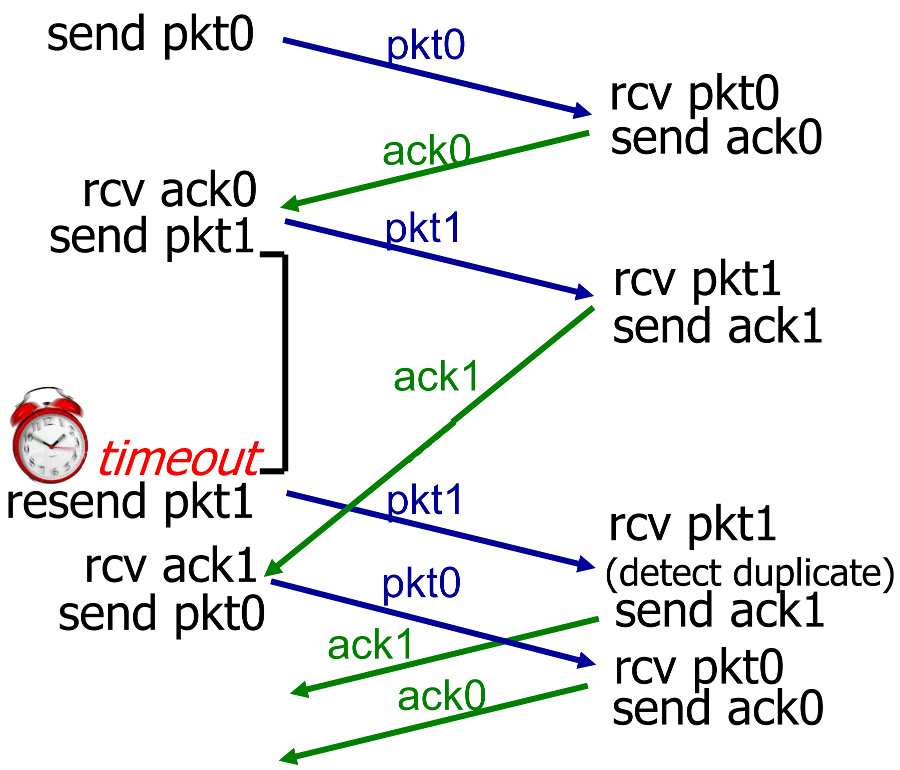
A simple stop-and-wait protocol



A simple stop-and-wait protocol



(c) ACK loss

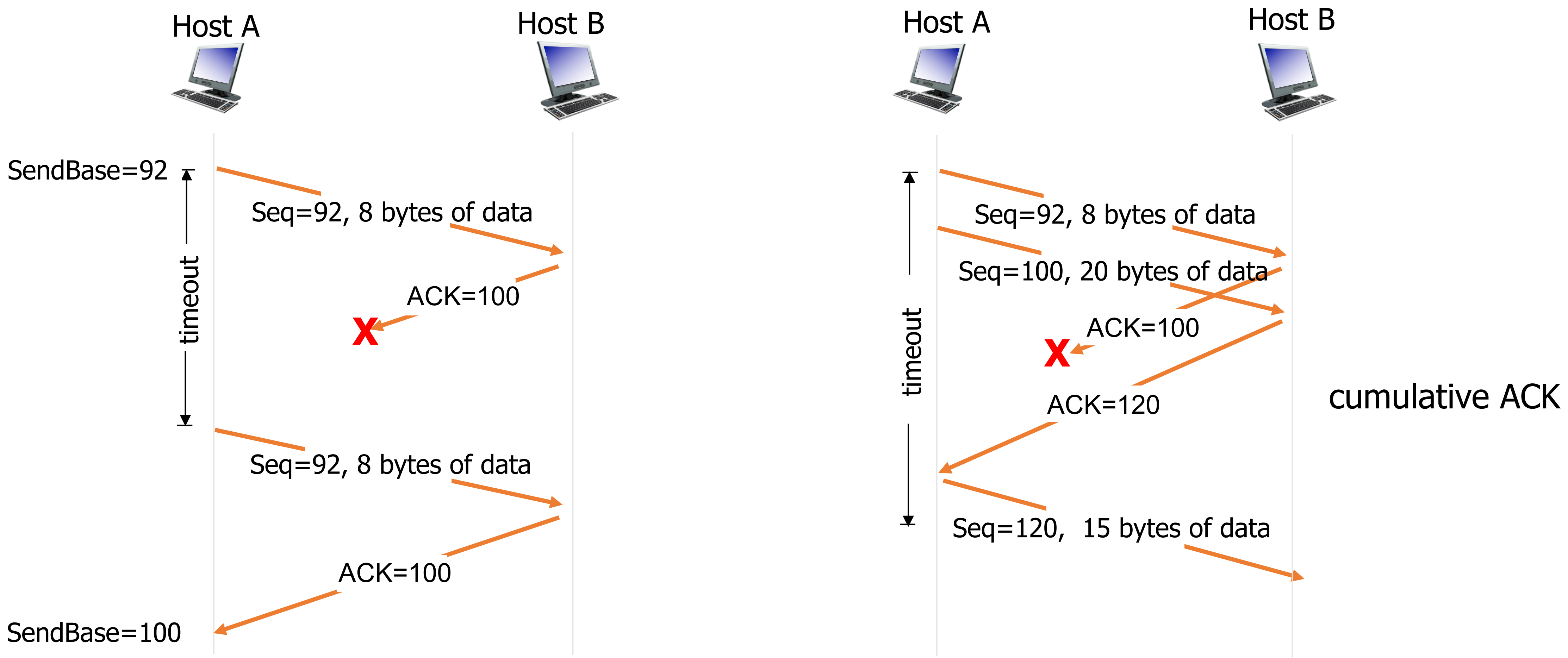


(d) premature timeout/ delayed ACK

TCP reliable data transfer

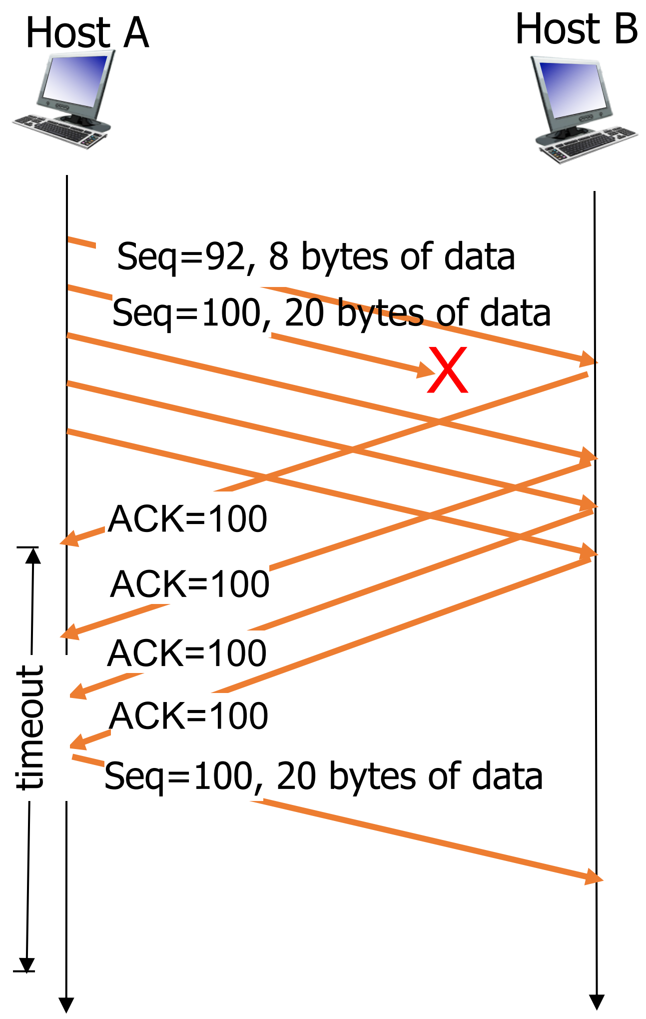
- TCP creates reliable data transfer service on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

TCP: retransmission scenarios



TCP fast retransmit

- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if a segment is lost, there will likely be many duplicate ACKs.
- *TCP fast retransmit*
  - if sender receives 3 duplicates ACKs for same data, resend unacked segment with smallest seq #
    - likely that unacked segment lost, so don't wait for timeout



Outline

Applications: HTTP (1.6, 5.2), DNS (13.2), …

Chapters 5 & 6 — RPC and RMI, indirect communication

Chapter 4 — Underlying interprocess communication primitives:

Sockets, message passing, multicast support, overlay networks

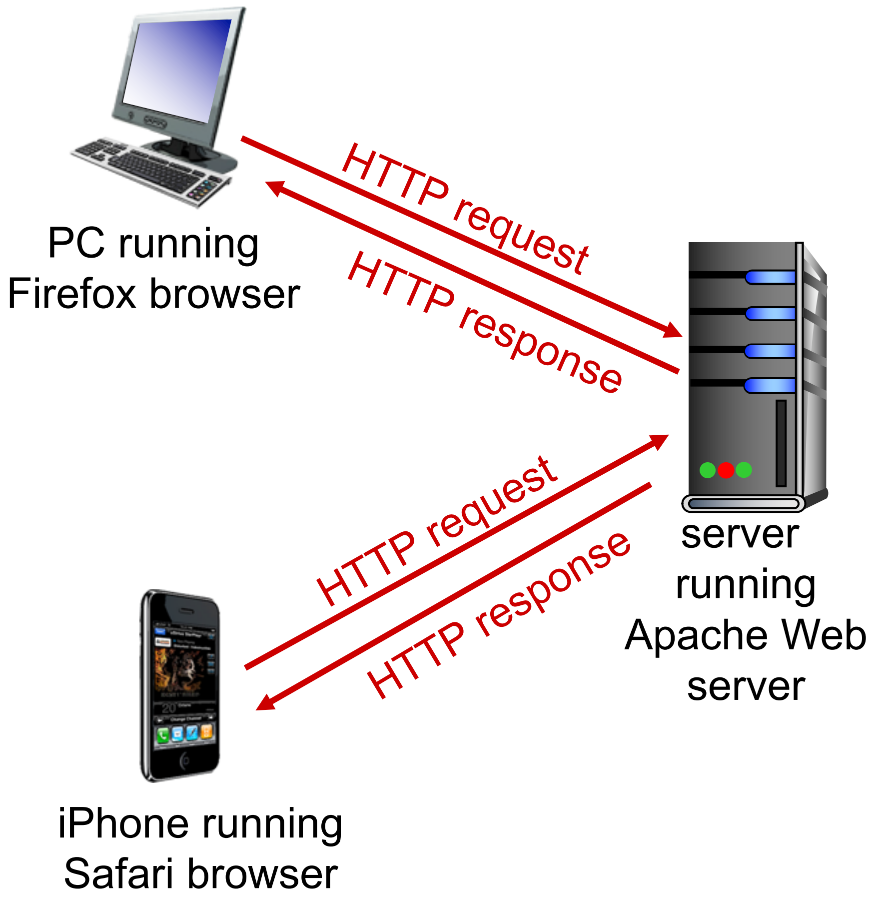
Chapter 3 — TCP/IP

Middleware layers

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
- RFC 2068, RFC 2616, RFC 7230



Non-persistent HTTP

suppose user enters URL: (contains text, references to 10 jpeg images)

`www.someSchool.edu/someDepartment/home.index`

1a. HTTP client initiates TCP connection to HTTP server (process) at www.someSchool.edu on port 80

1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80. "accepts" connection, notifying client

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object someDepartment/home.index

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

time

Non-persistent HTTP (cont.)

4. HTTP server closes TCP connection.

5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

time

6. Steps 1-5 repeated for each of 10 jpeg 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 (pipelining)

HTTP request message

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

carriage return character

line-feed character

request line (GET, POST, HEAD commands)

header lines

carriage return, line feed at start of line indicates end of header lines

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

HTTP response message

status line (protocol status code status phrase)

header lines

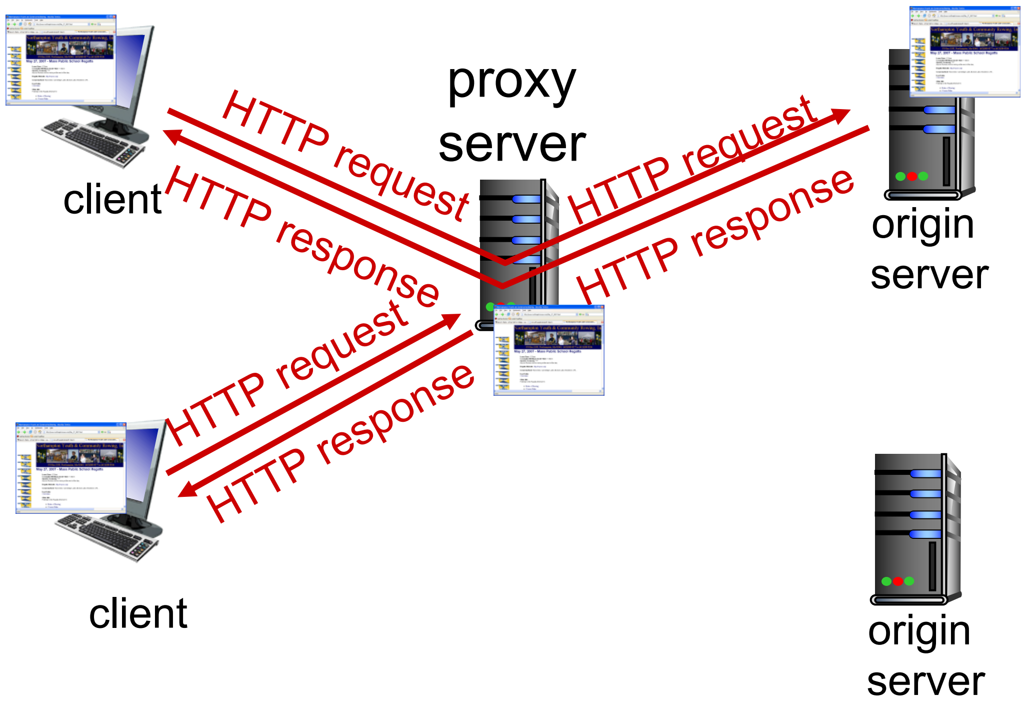
data, e.g., requested HTML file

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-1\r\n
\r\n
data data data data data ...
```

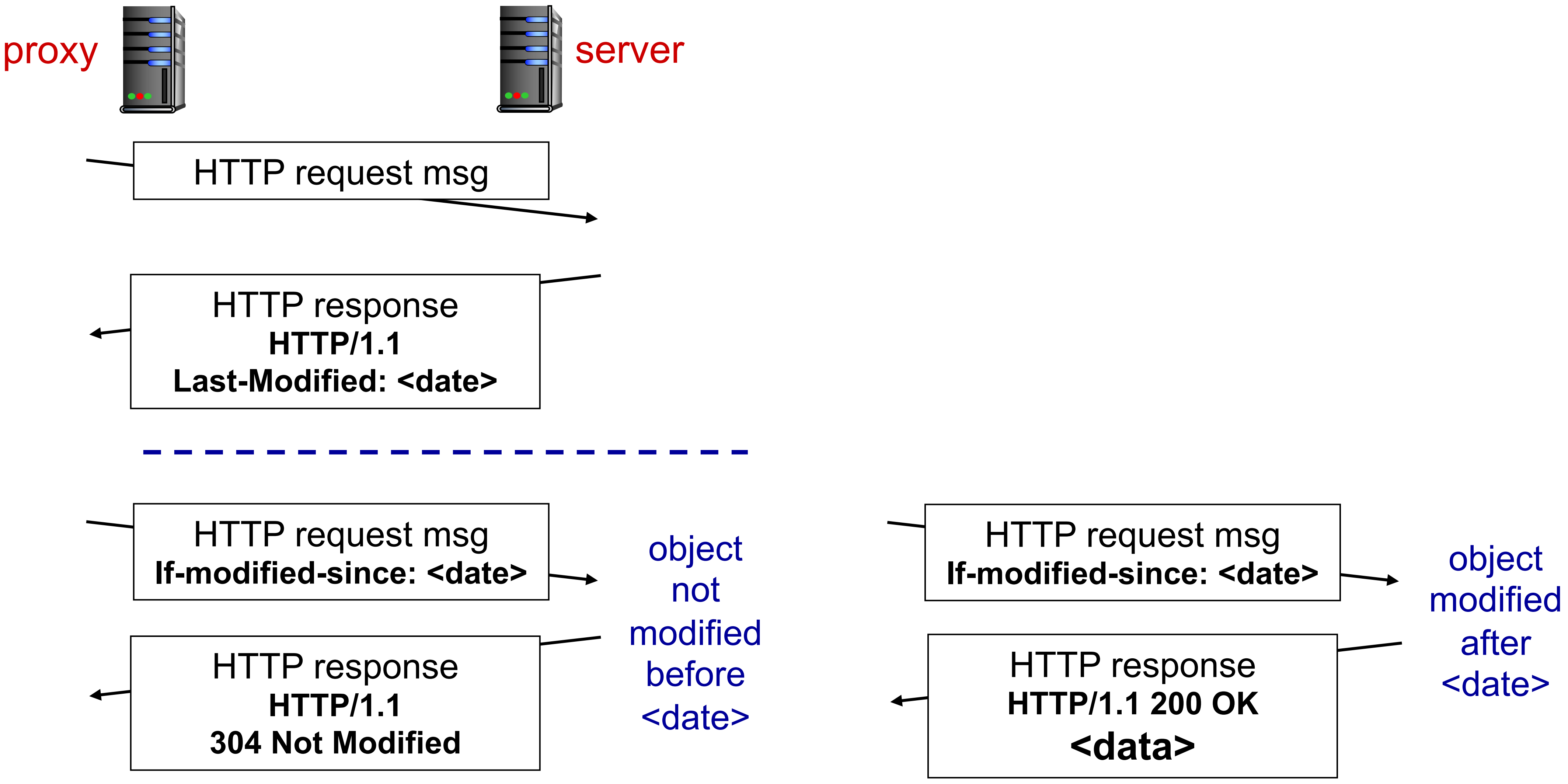
Web caches (proxy server)

- *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



Conditional GET



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
- reduce Internet traffic as a whole

DNS: domain name system

people: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

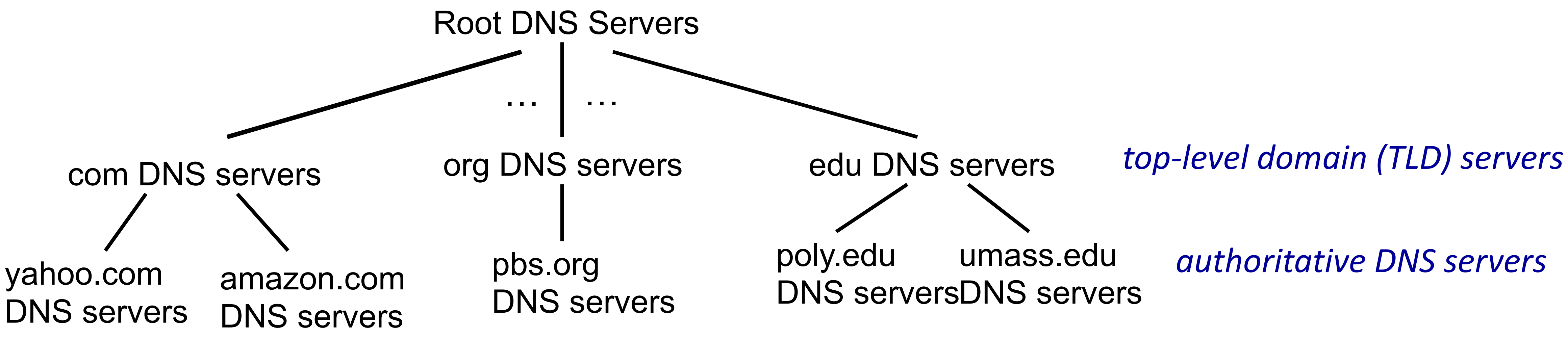
- IP address (32 bit) - used for addressing datagrams
- "name", e.g., www.yahoo.com - used by humans

Q: how to map between IP address and name, and vice versa ?

Domain Name System:

- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol:* hosts, name servers communicate to *resolve* names (address/name translation)
  - note: core Internet function, implemented as application-layer protocol
  - complexity at network's "edge"

DNS: a distributed, hierarchical database



client wants IP for www.amazon.com; 1st approximation:

- client queries root server to find com DNS server
- client queries .com DNS server to get amazon.com DNS server
- client queries amazon.com DNS server to get IP address for www.amazon.com

DNS: a distributed, hierarchical database

why not centralize DNS?

- single point of failure
- traffic volume
- distant centralized database
- maintenance: huge database, frequent update

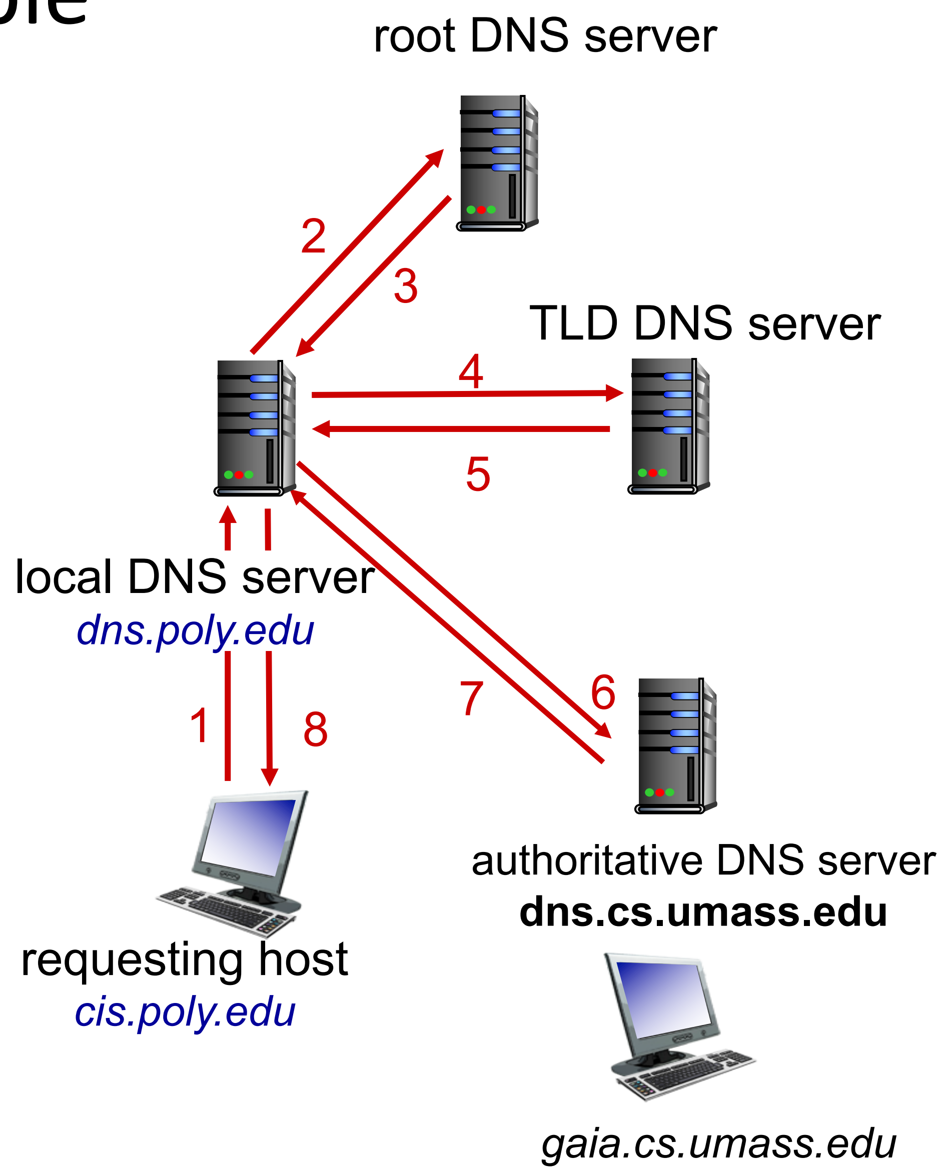
A: doesn't scale!

DNS name resolution example

- host at cis.poly.edu wants IP address for gaia.cs.umass.edu

iterative query:

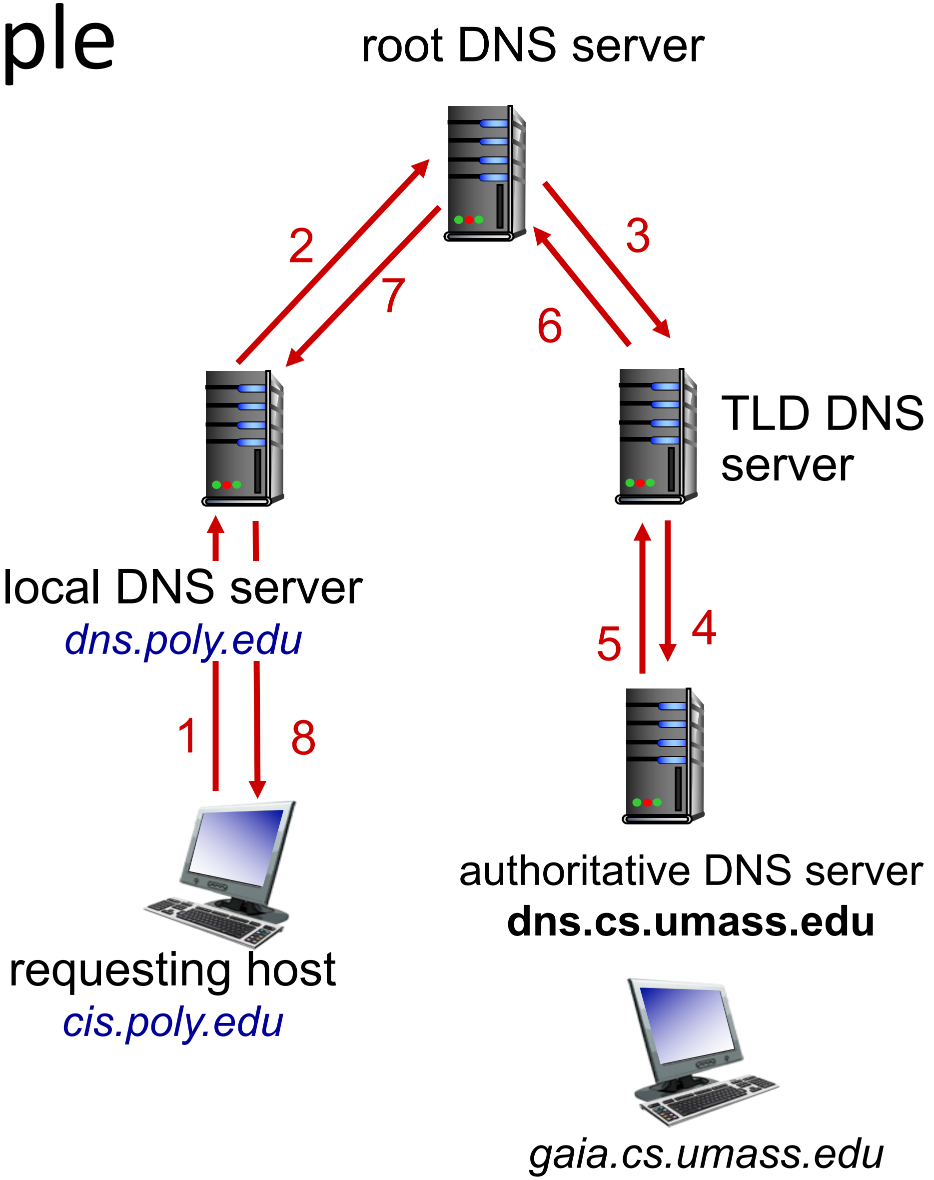
- contacted server replies with name of server to contact
- “I don’t know this name, but ask this server”
- All DNS query and replay messages are sent within UDP datagrams to port 53



DNS name resolution example

recursive query:

- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?



DNS: caching, updating records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time (Time to live, or TTL)
  - TLD servers typically cached in local name servers
    - thus root name servers not often visited
- cached entries may be *out-of-date* (best effort name-to-address translation!)
  - if name host changes IP address, may not be known Internet-wide until all TTLs expire
- update/notify mechanisms proposed IETF standard

DNS records

DNS: distributed database storing resource records (RR)

RR format: **(name, value, type, ttl)**

type=A

- **name** is hostname
- **value** is IP address

type=NS

- **name** is domain (e.g., foo.com)
- **value** is hostname of authoritative name server for this domain

type=CNAME

- **name** is alias name for some "canonical" (the real) name
- **www.ibm.com** is really **servereast.backup2.ibm.com**
- **value** is canonical name

type=MX

- **value** is canonical name of a mail server associated with alias **name**

Outline

Applications: HTTP (1.6, 5.2), DNS (13.2), …

Chapters 5 & 6 — RPC and RMI, indirect communication

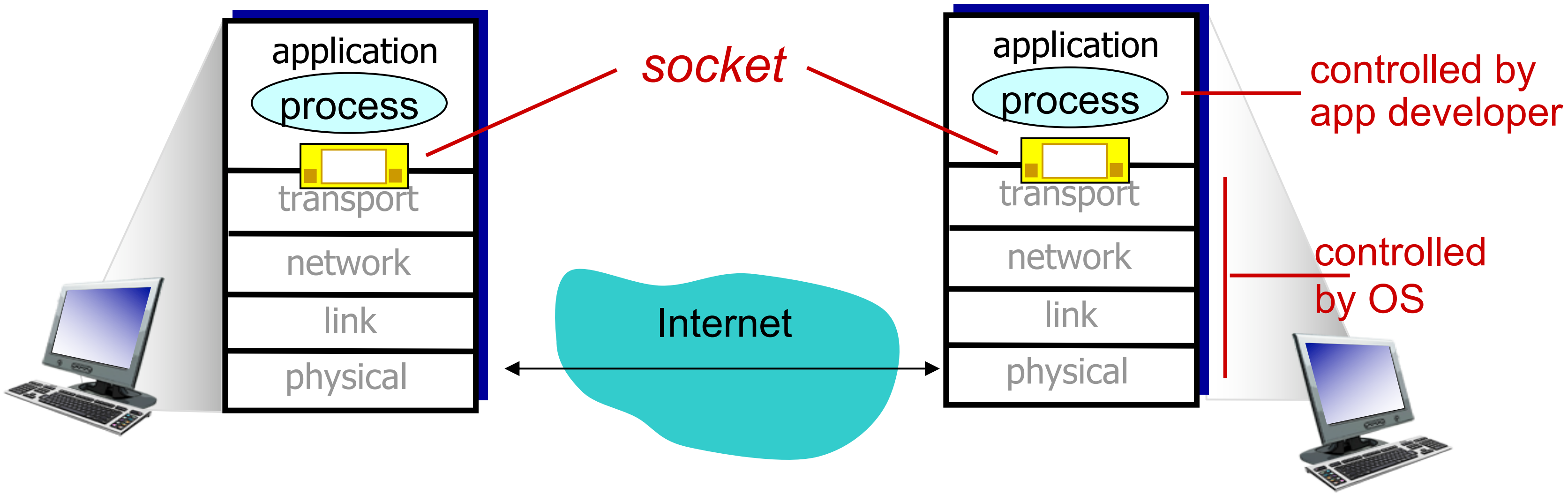
Chapter 4 — Underlying interprocess communication primitives:
Sockets, message passing, multicast support, overlay networks

Chapter 3 — TCP/IP

Middleware layers

Socket programming

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



Socket programming

Application Example:

1. client reads a string and sends it to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

Java Socket Programming: TCP client

```
import java.net.*;
import java.io.*;
public class TCPClient {
        public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
          try{
                int serverPort = 7896;
                s = new Socket(args[1], serverPort);
                DataInputStream in = new DataInputStream( s.getInputStream());
                DataOutputStream out =
                        new DataOutputStream( s.getOutputStream());
                out.writeUTF(args[0]);          // UTF is a string encoding see Sn 4.3
                String data = in.readUTF();
                System.out.println("Received: "+ data) ;
          }catch (UnknownHostException e){
                System.out.println("Sock:"+e.getMessage());
          }
          …
        }
}
```

Java Socket Programming: TCP server

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
          try{
                  int serverPort = 7896;
                  ServerSocket listenSocket = new ServerSocket(serverPort);
                  while(true) {
                          Socket clientSocket = listenSocket.accept();
                          Connection c = new Connection(clientSocket);
                  }
          } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
   }
}
```

```
class Connection extends Thread {
        DataInputStream in;
        DataOutputStream out;
        Socket clientSocket;
        public Connection (Socket aClientSocket) {
          try {
                clientSocket = aClientSocket;
                in = new DataInputStream( clientSocket.getInputStream());
                out =new DataOutputStream( clientSocket.getOutputStream());
                this.start();
           } catch(IOException e)  {System.out.println("Connection:"+e.getMessage());}
        }
        public void run(){
          try {                                   // an echo server
                String data = in.readUTF();
                out.writeUTF(data.toUpperCase());
          } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
          } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
          } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}}
        }
}
```

IP Multicast

- Unreliable multicast
  - UDP with multicast address: no guarantee on reliability and ordering
  - Reliable multicast discussed in Chapter 15
- Weak group membership service
  - Allow processes to join or leave groups dynamically
  - Does not maintain group views
  - View-synchronous group communication discussed in Chapter 18

- IP multicast addresses: 224.0.0.0 - 239.255.255.255
- IP broadcast address: 255.255.255.255

An Example of Java IP multicast

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
        public static void main(String args[]){
         // args give message contents & destination multicast group (e.g., "228.5.6.7")
        MulticastSocket s =null;
         try {
                InetAddress group = InetAddress.getByName(args[1]);
                s = new MulticastSocket(6789);
                s.joinGroup(group);
                byte [] m = args[0].getBytes();
                DatagramPacket messageOut =
                        new DatagramPacket(m, m.length, group, 6789);
                s.send(messageOut);
```

Any UDP socket can send to multicast addresses. However, to receive multicast datagrams, you must join that specific group address.

```
        // get messages from others in group
                byte[] buffer = new byte[1000];
                for(int i=0; i< 3; i++) {
                    DatagramPacket messageIn =
                            new DatagramPacket(buffer, buffer.length);
                    s.receive(messageIn);
                    System.out.println("Received:" + new String(messageIn.getData()));
                }
                s.leaveGroup(group);
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
       }catch (IOException e){System.out.println("IO: " + e.getMessage());}
     }finally {if(s != null) s.close();}
  }
}
```

Outline

Applications: HTTP (1.6, 5.2), DNS (13.2), …

Chapters 5 & 6 — RPC and RMI, indirect communication

Chapter 4 — Underlying interprocess communication primitives:

Sockets, message passing, multicast support, overlay networks

Chapter 3 — TCP/IP

Middleware layers

Remote Procedure Call

- Data representation and marshalling
- Programming with interfaces
- Supporting different call semantics
- Providing at least location & access transparencies

External data representation and marshalling

- Heterogeneity in data representation
  - integers: Big-endian vs. little-endian
  - characters: ASCII vs. Unicode
  - floating-point numbers, arrays, structures, objects, …

External data representation and marshalling

- A common data representation
  - Data converted to an agreed extern format
  - Data transmitted in sender's format together with an indicator of format
- Marshalling - taking a collection of data items and assembling them into a form suitable for transmission in a message
- Unmarshalling – disassembling data on arrival on arrival to produce an equivalent collection of data items at the destination

Examples of Data Representation Approaches

- CORBA's common data representation
  - External representation for primitive and structured types
  - Support a variety of languages
- Java object serialization
  - Flattening and representation of any single object or a tree of objects
  - Java only
- XML
  - A textual format for representing structured data
  - May refer to externally defined namespaces
- More recent approaches: Google's protocol buffers, JSON, …

Java Object Serialization

- *Person p = new Person(1984 , "Smith", "London");*

Serialized values				Explanation
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name	java.lang.String place	*number, type and name of instance variables*
1984	5 Smith	6 London	h1	*values of instance variables*

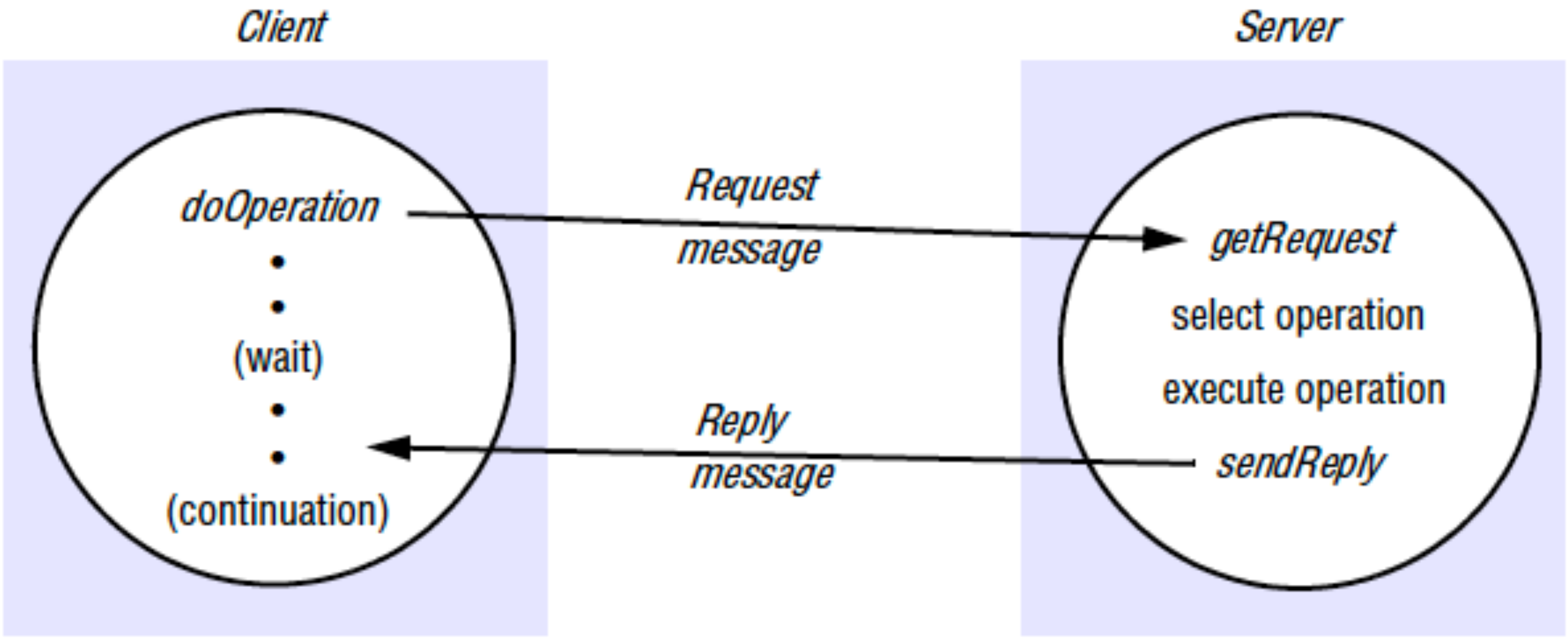
The true serialized form contains additional type markers; h0 and h1 are handles

- Implemented by java.io.ObjectOutputStream and java.io.ObjectInputStream
- Not everything should be serialized: e.g., references to local files

Programming with interfaces

- Separation between interface and implementation details
- Manage heterogeneity in programming languages and platforms
- Support for software evolution

Request-reply communication



Operations of the request-reply protocol

public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)
Sends a request message to the remote server and returns the reply.
The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

public byte[] getRequest ();
Acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
Sends the reply message *reply* to the client at its Internet address and port.

Request-reply message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*// array of bytes*

- A message identifier includes a *requestId* and an identifier for the sender process (e.g., sender's IP address and port number)

Common Failure Types

- Omission failures: request or reply message lost
- Crash failures: server crashes (before or after the procedure is executed)
- Byzantine failures

Call Semantics

Fault tolerance measures			Call semantics
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	

Call Semantics

Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*

- Maybe semantics: the remote procedure call may be executed once or not at all

Call Semantics

Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*

- At-least-once semantics: the invoker receives either a result, in which case the invoker knows that the procedure was executed at least once, or an exception informing it that no result was received.
- Idempotent operation: performed repeatedly has the same effect as performed exactly once

Call Semantics

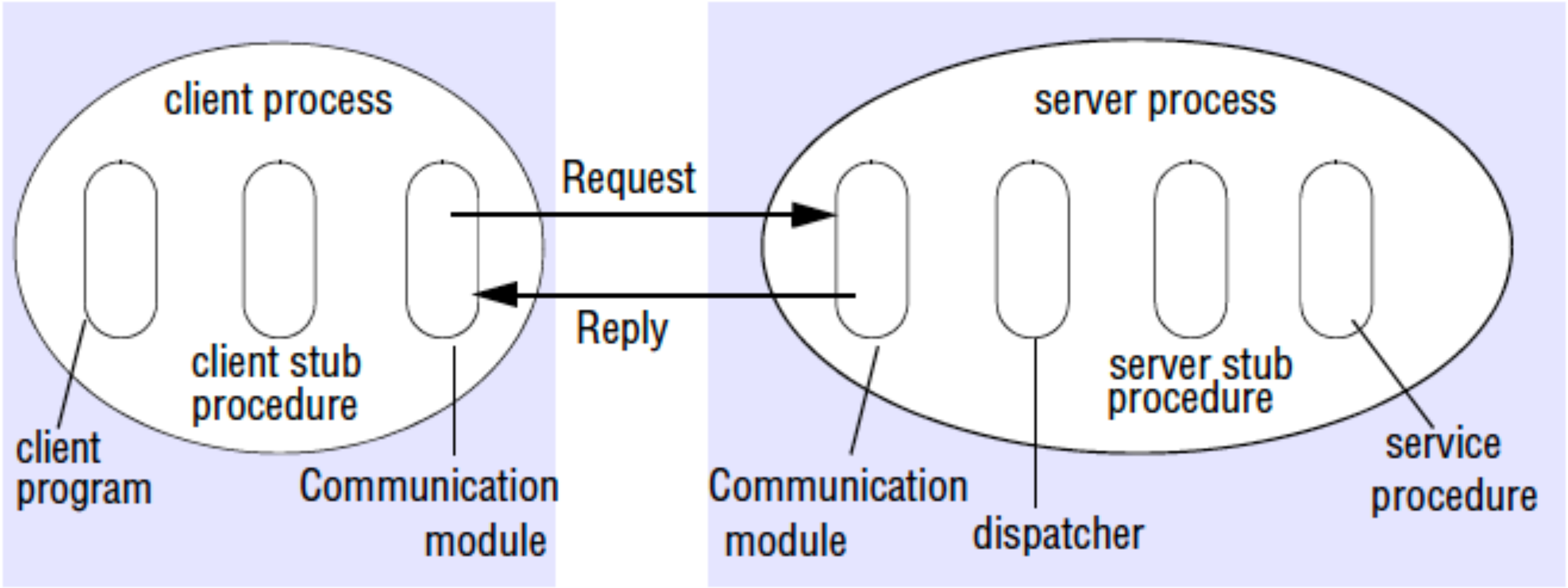
Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

- At-most-once semantics: the caller receives either a result, in which case the caller knows that the procedure was executed exactly once, or an exception informing it that no result was received, in which case the procedure will have been executed either once or not at all.

Transparency

- Ideally, RPC should provide at least location & access transparencies
- In practice, RPC needs to deal with
  - failures of the network and remote server process: hard to distinguish
  - latency: abort a remote call that takes too long (restore things at the server)
  - call by value only
- Current consensus
  - Provide same syntax to local and remote calls
  - Expose the differences at the service interface: remote exception, call semantics, etc.

Remote Procedure Call: Implementation

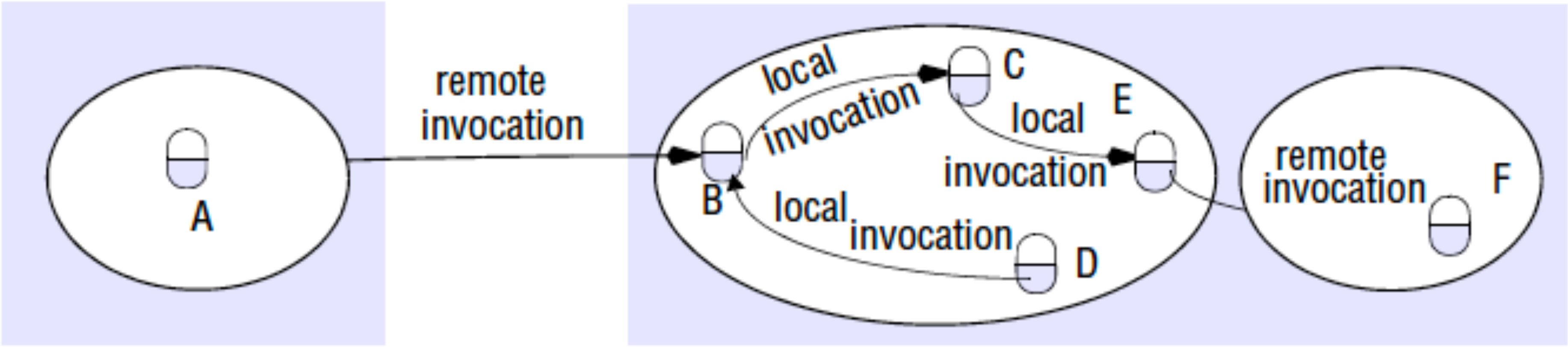


- Stub: marshalling and unmarshalling
- Communication module: request-reply, call semantics

From RPC to RMI

- Commonalities:
  - Programming by interfaces
  - Similar call semantics
  - Similar level of transparency
- Differences
  - RMI provides full expressive power of object-orient programming in distributed settings
  - All objects in RMI (local or remote) have unique object refences: call by reference

Remote and local method invocations



- local method invocations: method invocations between objects in the same process
- remote method invocations: method invocations between objects in different processes, whether in the same computer or not
- remote objects: objects that can receive remote invocations

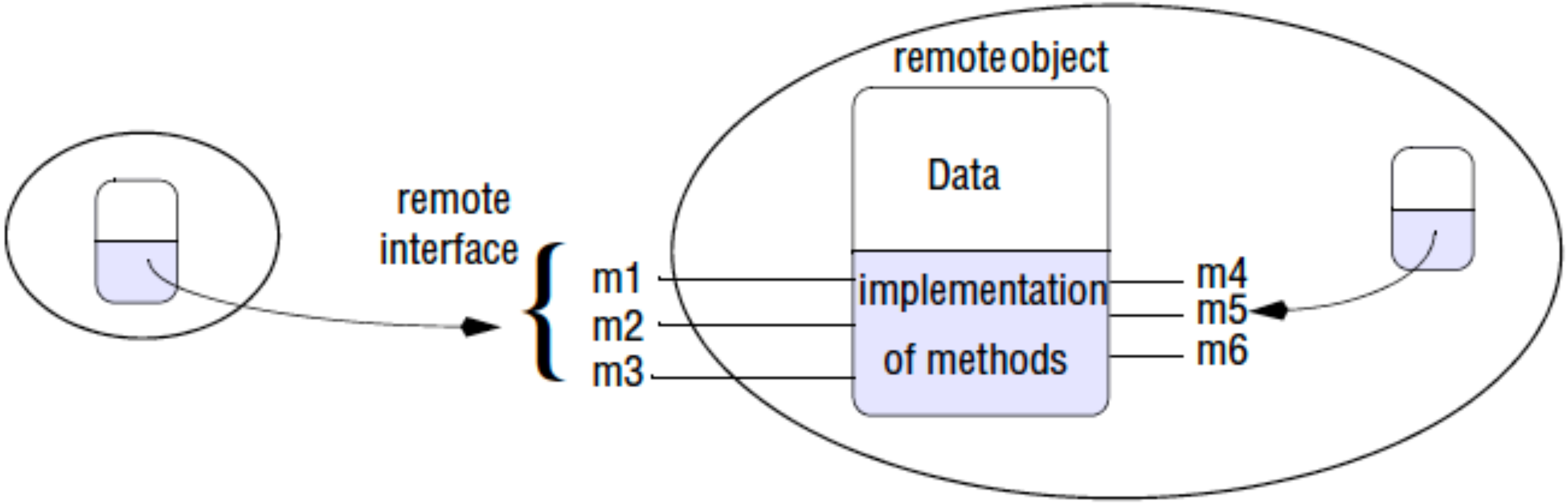
Remote Object References

- Other objects can invoke the methods of a remote object if they have access to its remote object reference
- Remote object references may be passed as arguments and results of remote method invocations
- Each remote object has a unique remote object reference
- Example:

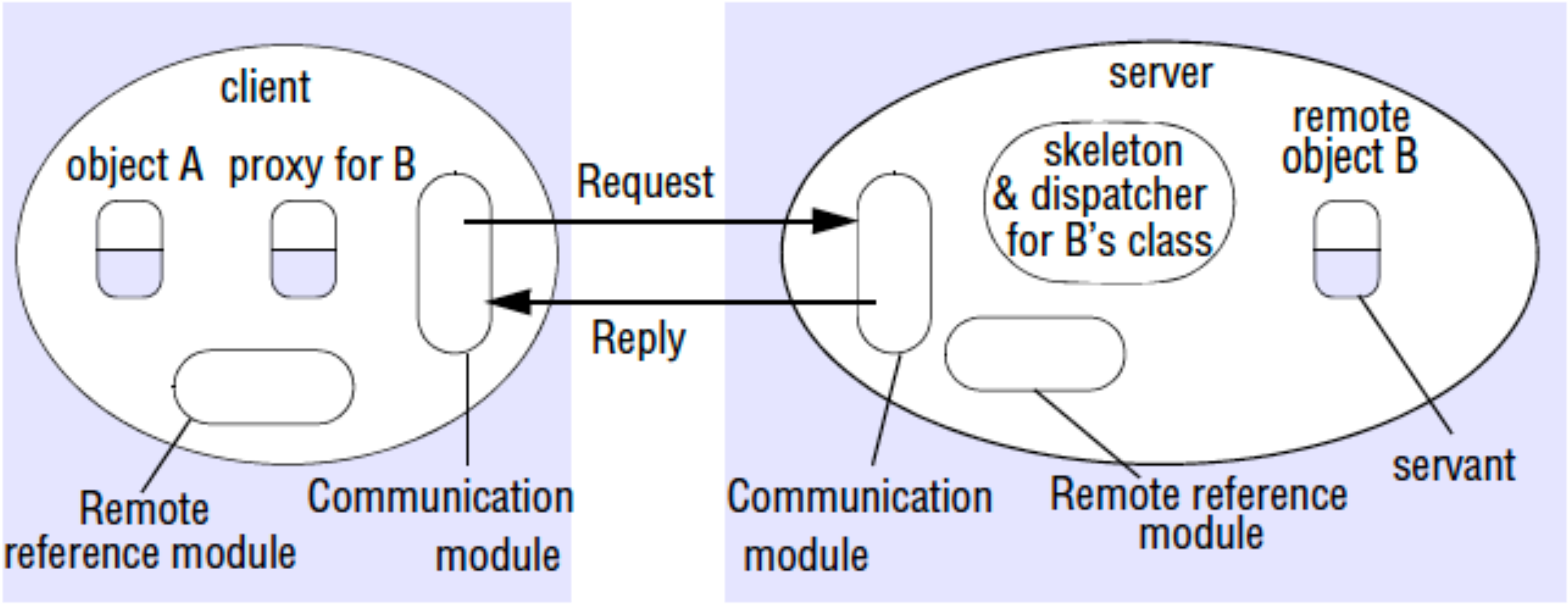
32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

  - Not location transparent

A remote object and its remote interface



Implementation of RMI



- The classes for the proxy, dispatcher and skeleton are generated automatically by an interface compiler

Case Study: Java RMI

- Built on top of TCP
- Generic dispatcher via reflection (since Java 1.2)
  - No skeleton needed
- Dynamic stub generation (since J2SE 5.0)
- Dynamic class downloading: java codebase
- Distributed garbage collection
- Activatable objects

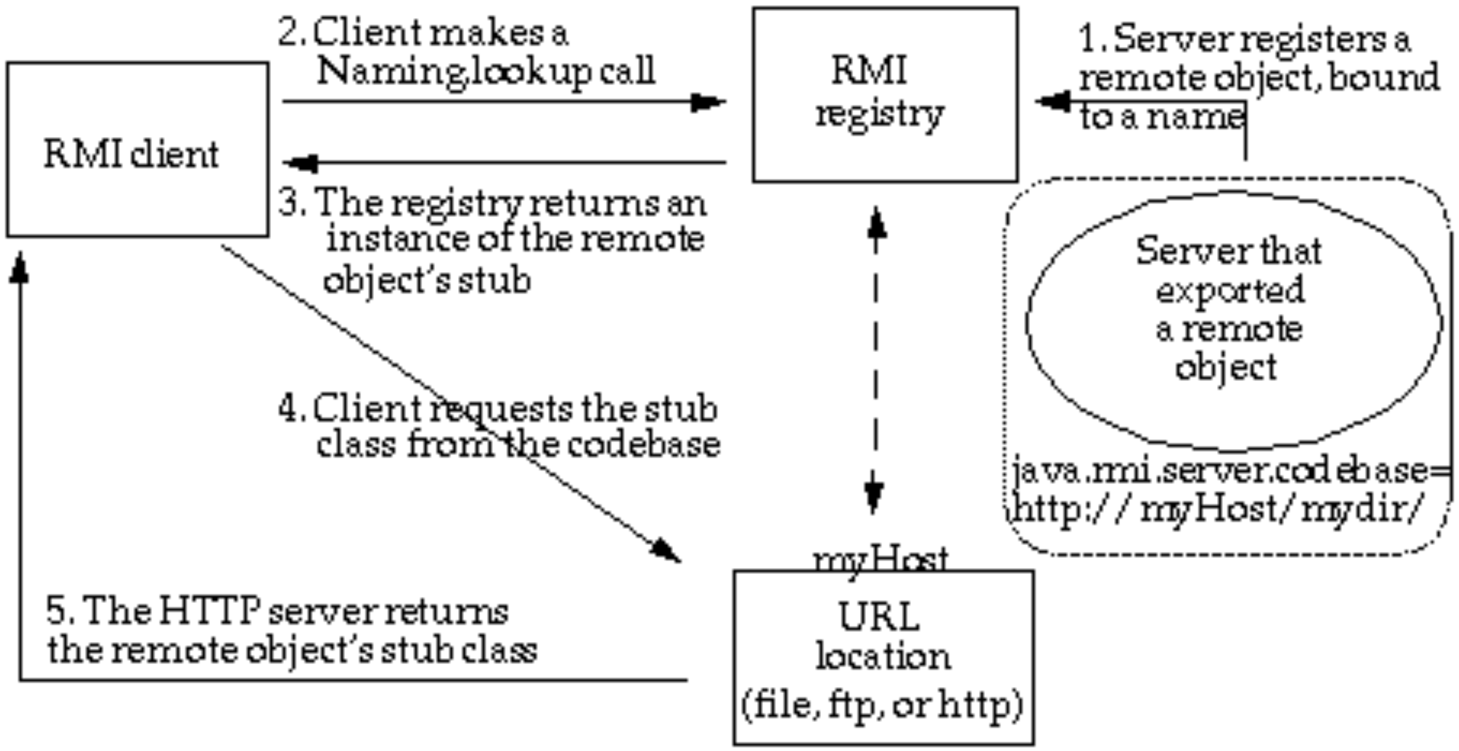
Case Study: Java RMI

- Remote interfaces
  - defined by extending the *Remote* interface in *java.rmi*
  - the methods must throw *RemoteExeption*
- Parameter and result passing
  - Only passing objects that are serializable (implementing the *Serializable* interface)
  - Remote objects are passed by reference
  - Non-remote objects are passed by value

The Binder - RMIregistry

- Client programs generally require a means of obtaining a remote object reference for at least one of the remote objects held by a server
- A binder is a naming service that maintains a table containing mappings from textual names to remote object references
  - used by servers to register their remote objects by name and by clients to look them up

RMIregistry and Codebase



https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/codebase.html

Example: “Hello World” with Java RMI

- https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/hello/hello-world.html

  Hello.java - a remote interface

  Server.java - a remote object implementation that implements the remote interface

  Client.java - a simple client that invokes a method of the remote interface

Hello.java

```
package example.hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
	String sayHello() throws RemoteException;
}
```

Server.java

```
package example.hello;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server implements Hello {
    public Server() {}
    public String sayHello() { return "Hello, world!"; }
    public static void main(String args[]) {
        try {
            Server obj = new Server();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
            // Bind the remote object's stub in the registry
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Hello", stub);
            System.err.println("Server ready");
        }   catch (Exception e) {  System.err.println("Server exception: " + e.toString()); e.printStackTrace(); }
    }
}
```

using an anonymous port

Client.java

```
package example.hello;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
    private Client() {}
    public static void main(String[] args) {
        String host = (args.length < 1) ? null : args[0];
        try {
            Registry registry = LocateRegistry.getRegistry(host);
            Hello stub = (Hello) registry.lookup("Hello");
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) { System.err.println("Client exception: " + e.toString()); e.printStackTrace(); }
    }
}
```

To run the example

Compile the source files: javac -d ***destDir*** Hello.java Server.java Client.java

Start the Java RMI registry: rmiregistry & (mac, linux)

start rmiregistry (windows)

Start the server:

java -classpath ***classDir*** -Djava.rmi.server.codebase=file:***classDir***/ example.hello.Server &

Run the client:

java -classpath ***classDir*** example.hello.Client

Lab 1: Implementing an Election service using Java RMI

- Election interface
  - vote (string name, int voter)
  - result (string name, int num)
- Server: implement the Election service
- Clients: submit votes and query result
- Requirements
  - ensure each user votes once only
  - records remain consistent when accessed concurrently by multiple clients
    - e.g., using synchronized methods
  - records are safely stored even when the server process crashes
    - e.g., saving records to a file on your disk