COMP2211: Networks and Systems

Networks: Application Layer

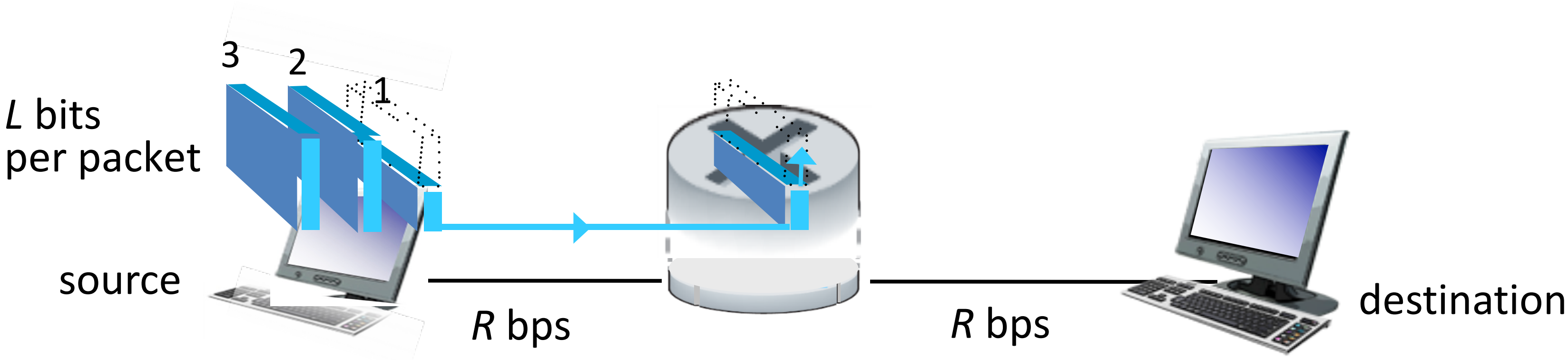
Donald Sturgeon
donald.j.sturgeon@durham.ac.uk

Department of Computer Science

Previously

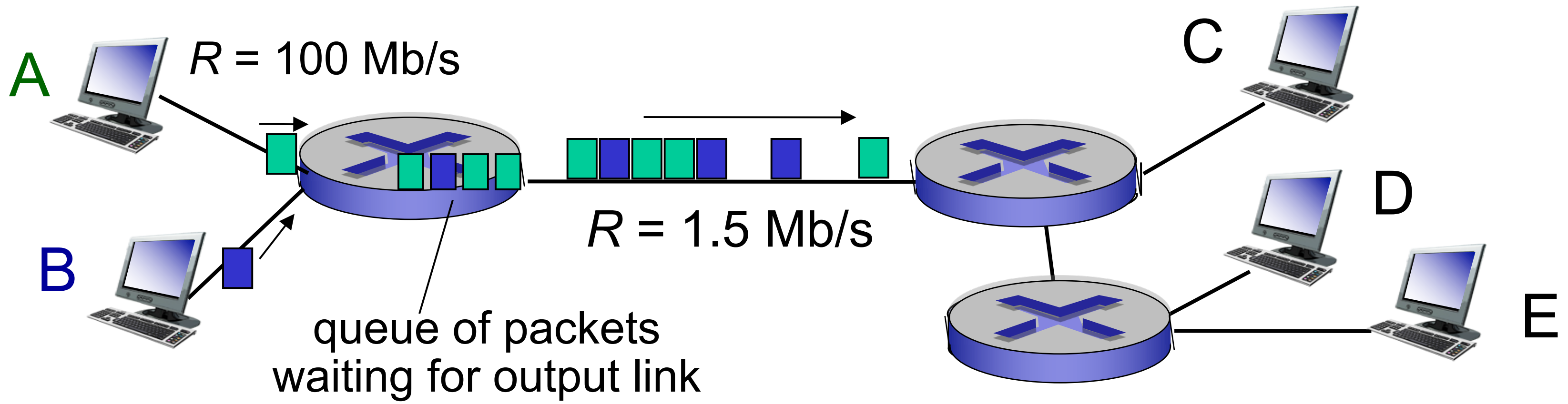
- Network core
  - Packet switching
  - Circuit switching
- Delay and loss in networks
- Protocol layers

Packet-switching: store-and-forward



- *Store and forward:* entire packet must arrive at router before it can be transmitted on next link
- Takes *L*/*R* seconds to transmit (push out) *L*-bit packet into link at *R* bps

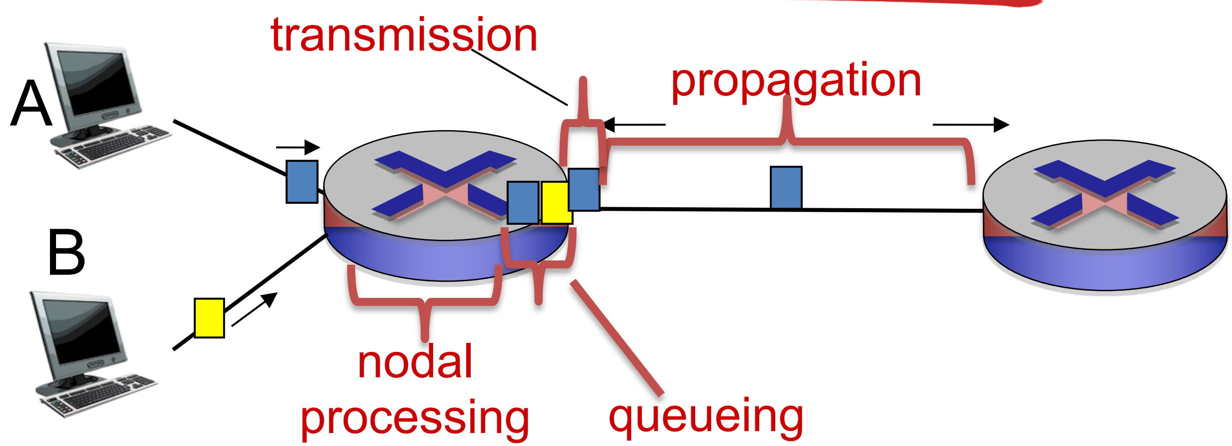
Packet Switching: queuing delay, loss



Queuing and loss:

- If arrival rate (in bits) to link exceeds transmission rate of link for a period of time:
  - Packets will queue, wait to be transmitted on link.
  - Packets can be dropped (lost) if memory (buffer) fills up.

Four sources of packet delay



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

d_{proc}: nodal processing
- Check bit errors
- Determine output link
- Typically < msec

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
- $d_{prop} = d/s$

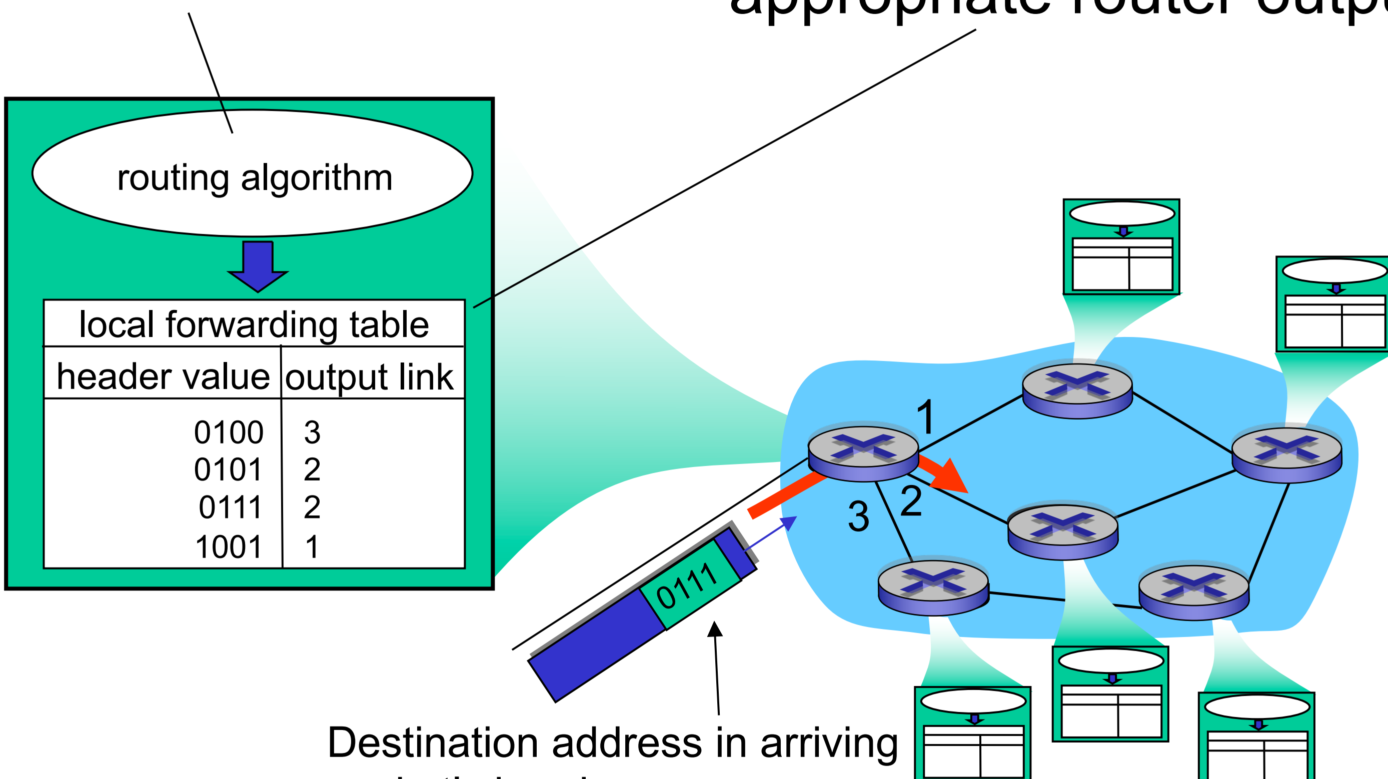
$$\boxed{d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop}}$$

Two key network-core functions

Routing: determines source-destination route taken by packets

- *routing algorithms*

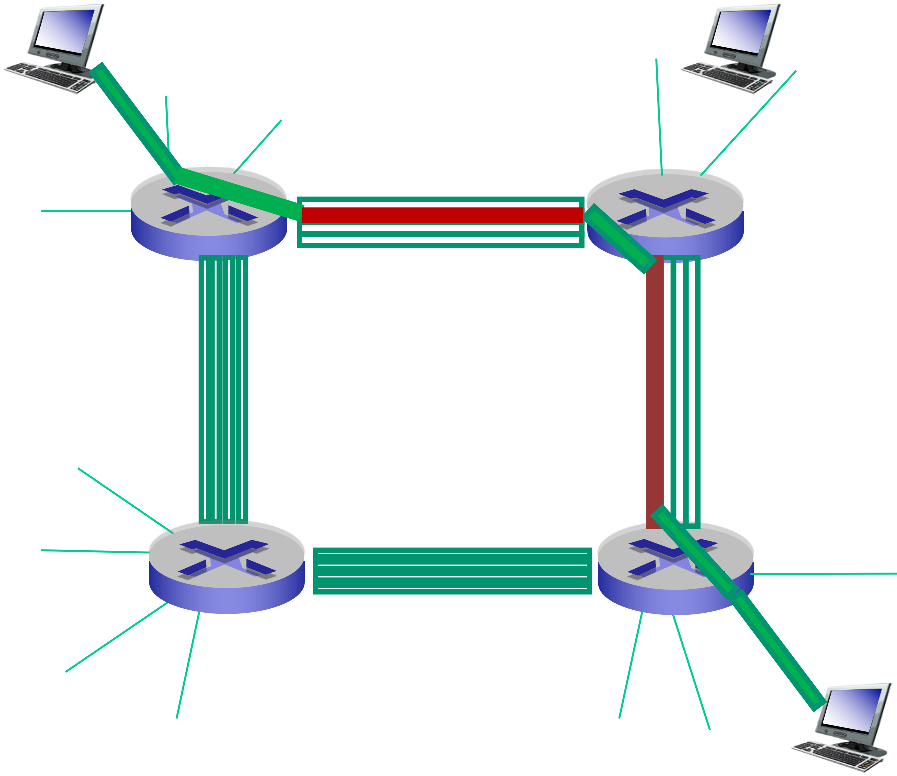
Forwarding: move packets from router's input to appropriate router output



Alternative core: circuit switching

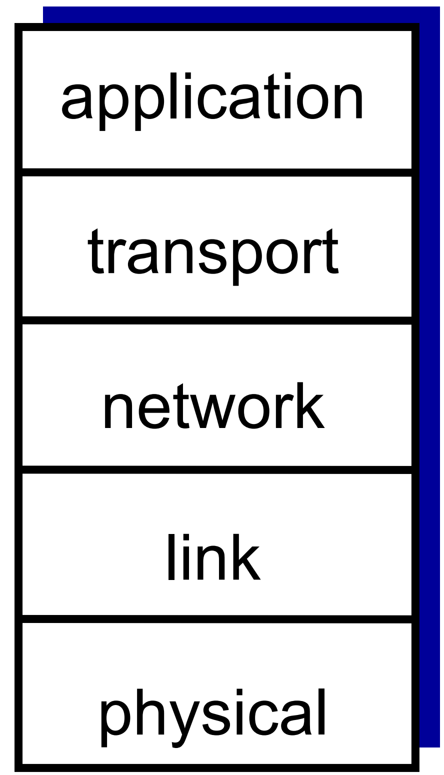
End-end resources allocated to, reserved for "call" between source & dest:

- In diagram, each link has four circuits.
  - Call gets 2nd circuit in top link and 1st circuit in right link.
- Dedicated resources: no sharing
  - Circuit-like (guaranteed) performance
- Circuit segment idle if not used by call *(no sharing)*
- Commonly used in traditional telephone networks



Internet protocol stack

- *Application:* supporting network applications
  - FTP, SMTP, HTTP
- *Transport:* process-process data transfer
  - TCP, UDP
- *Network:* routing of datagrams from source to destination
  - IP, routing protocols
- *Link:* data transfer between neighboring network elements
  - Ethernet, 802.11 (WiFi), PPP
- *Physical:* bits "on the wire"



Application Layer

- Network architecture:
  - Client-server architecture
  - P2P architecture

- Protocols:
  - TCP (Transmission Control Protocol)
  - UDP (User Datagram Protocol)

- Socket programming:
  - TCP
  - UDP

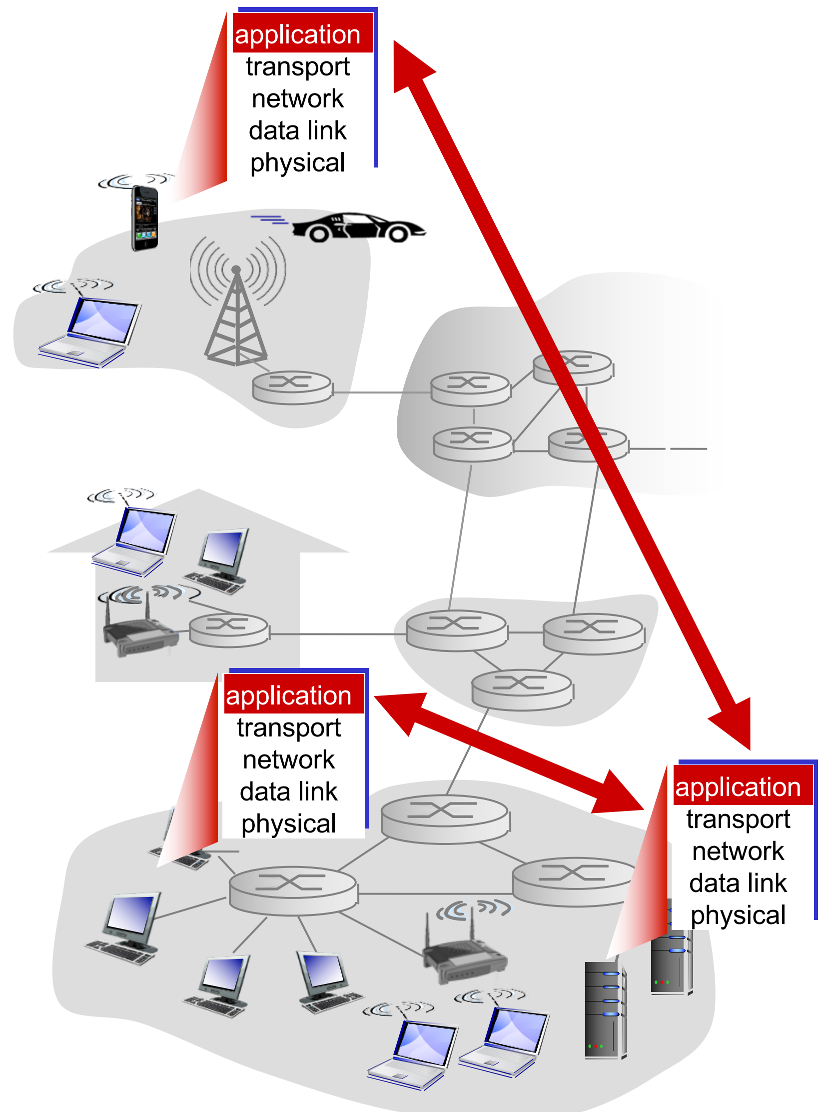
Creating a network app

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 allow for rapid app development, propagation

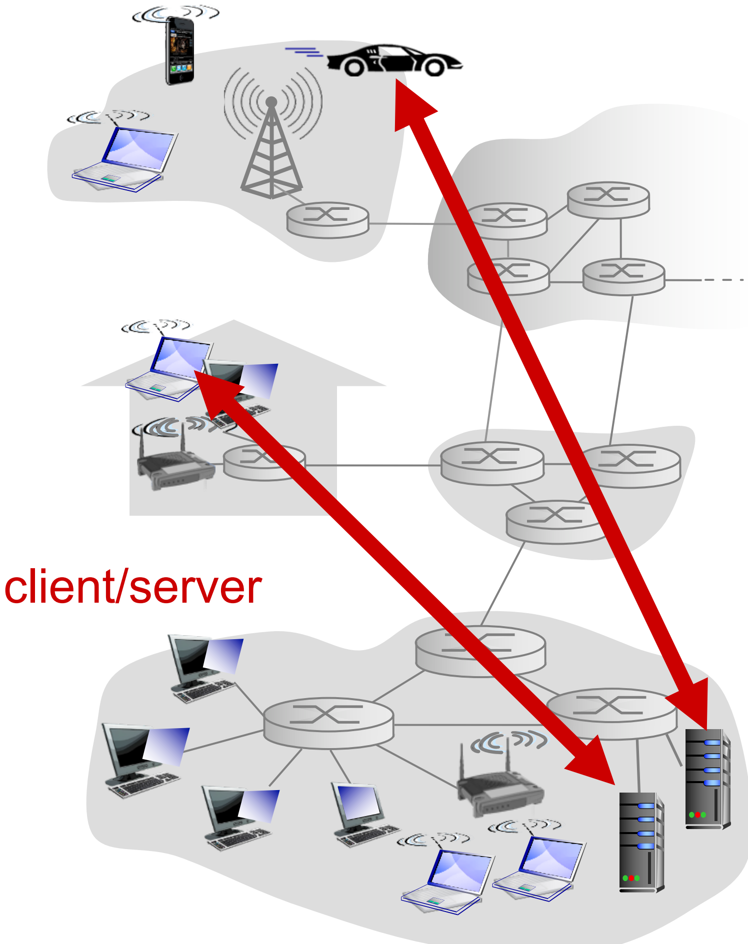


Application architectures

Possible structures of applications:

- Client-server
- Peer-to-peer (P2P)

Client-server architecture



Server:

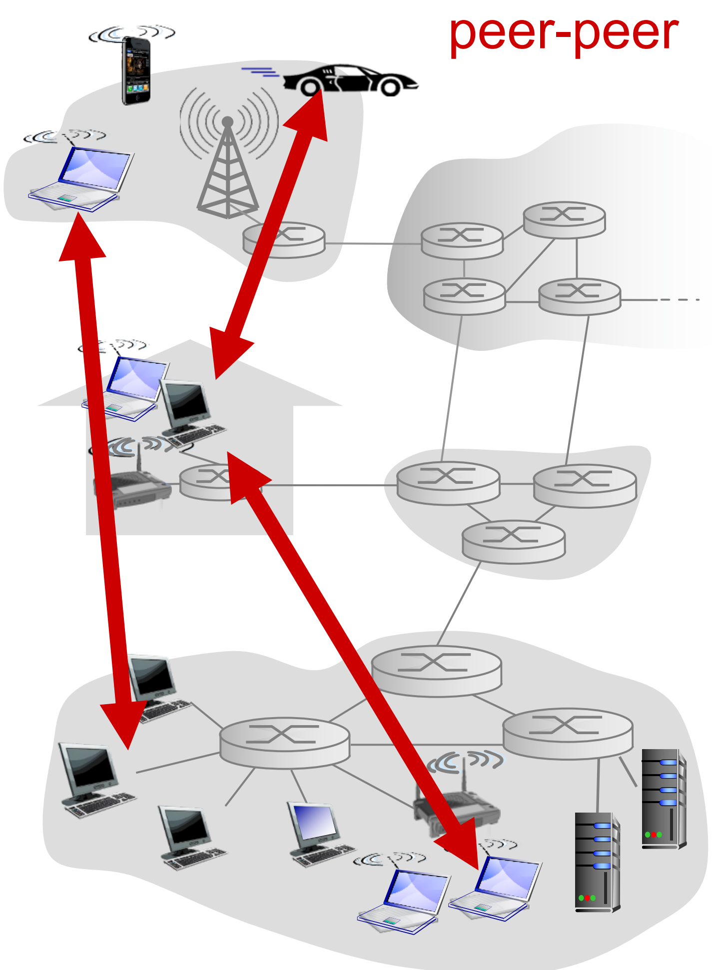
- Always-on host
- Fixed (static) IP address
- Data centres for scaling

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
- 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



Road map

- Network architecture:
  - Client-server architecture
  - P2P architecture
- Processes:
  - TCP
  - UDP
- Web and HTTP
- Socket programming:
  - TCP
  - UDP

Processes communicating

Process: program running within a host

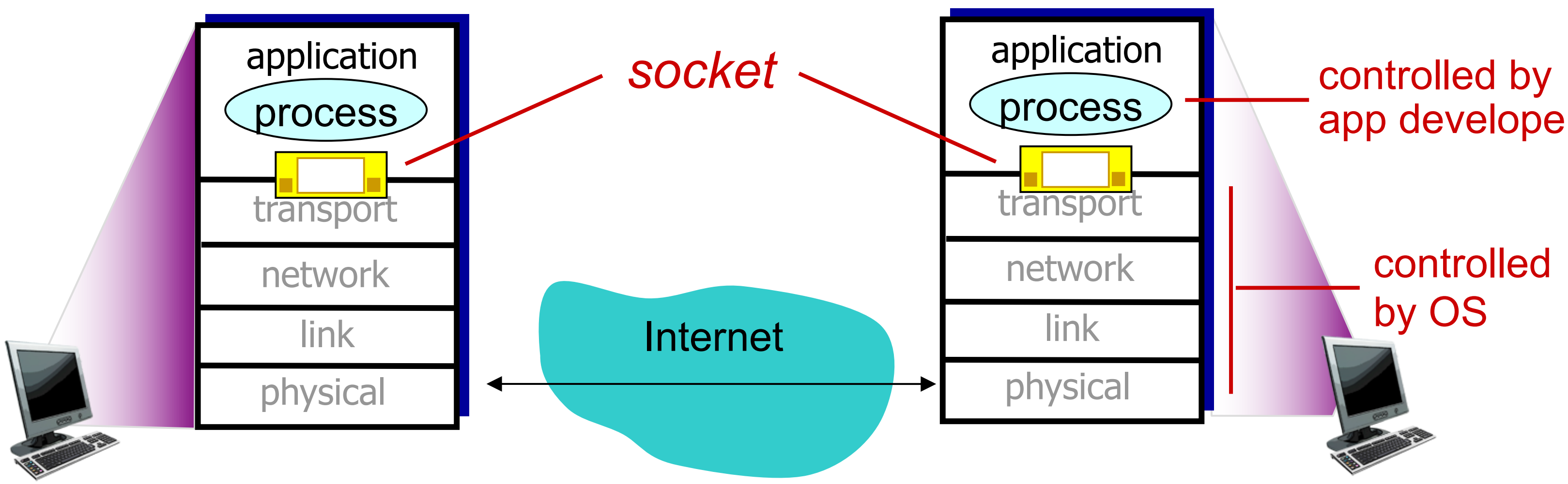
- Within same host, two processes communicate using inter-process communication (defined by OS)
- Processes in different hosts communicate by exchanging messages

Socket: a software mechanism that allows a process to create and send messages into, and receive messages from the network.

- A process is analogous to a house, and its socket is analogous to its door
- Interface between application layer and transport layer.

Sockets

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



What transport service does an app need?

Durham University

Data integrity

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

Security

- Encryption, data integrity, …

Timing

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

Internet transport protocols services

TCP service:

- *Connection-oriented:* setup required between client and server processes
- *Reliable transport* between sending and receiving process
- *Flow control:* sender won't overwhelm receiver
- *Full-duplex connection:* connection can send messages to each other at the same time

UDP service:

- *Unreliable data transfer* between sending and receiving processes
- *Does not provide:* reliability, flow control, congestion control, timing, security, or connection setup

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

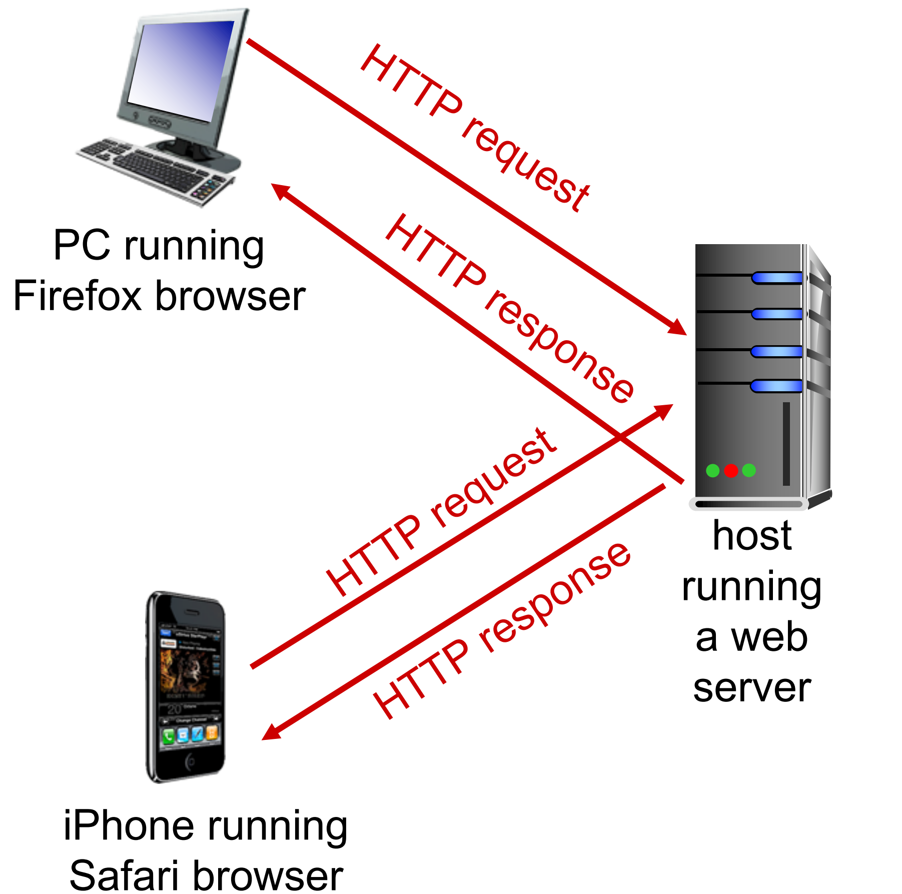
Open protocols:
Defined in Request For Comments (RFC)
Allow for interoperability, e.g. HTTP, SMTP

Proprietary protocols
e.g. Skype

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)

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 CS-department/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. But, not really! (until the client received the response intact)

5. HTTP client receives response message containing html file, displays html. Parsing html file, finds ***n*** referenced objects (e.g., JPEG)

time

6. First 4 steps are repeated for each of 10 jpeg objects

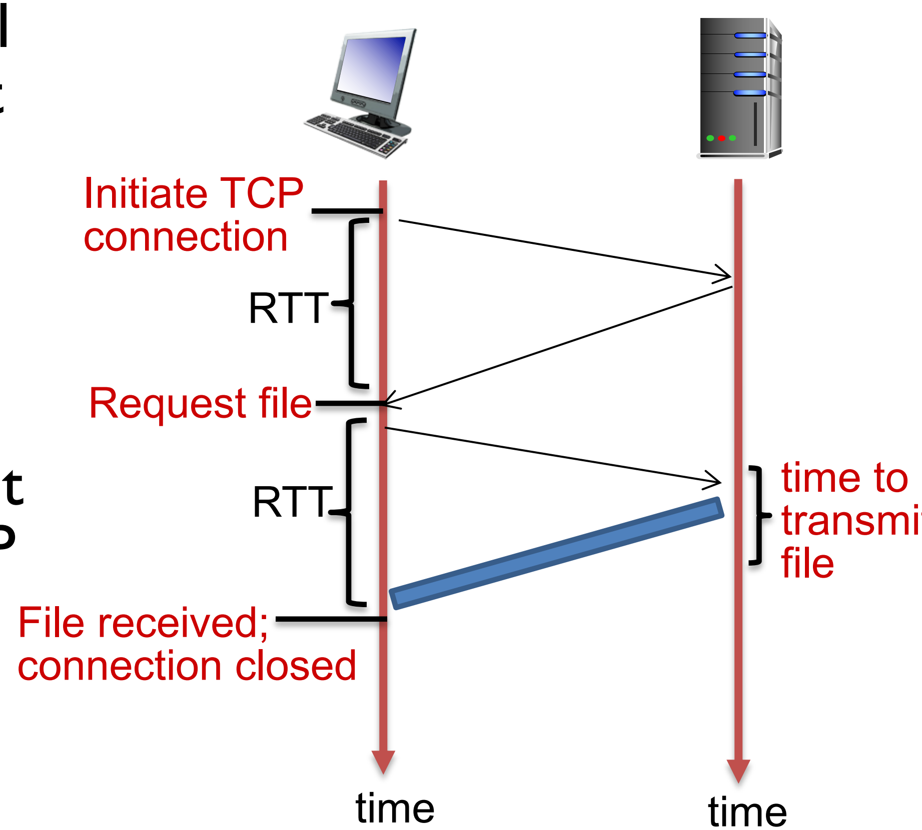
Non-persistent HTTP: response time

RTT (round-trip time) (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

  Incurred for *each* file



Persistent HTTP

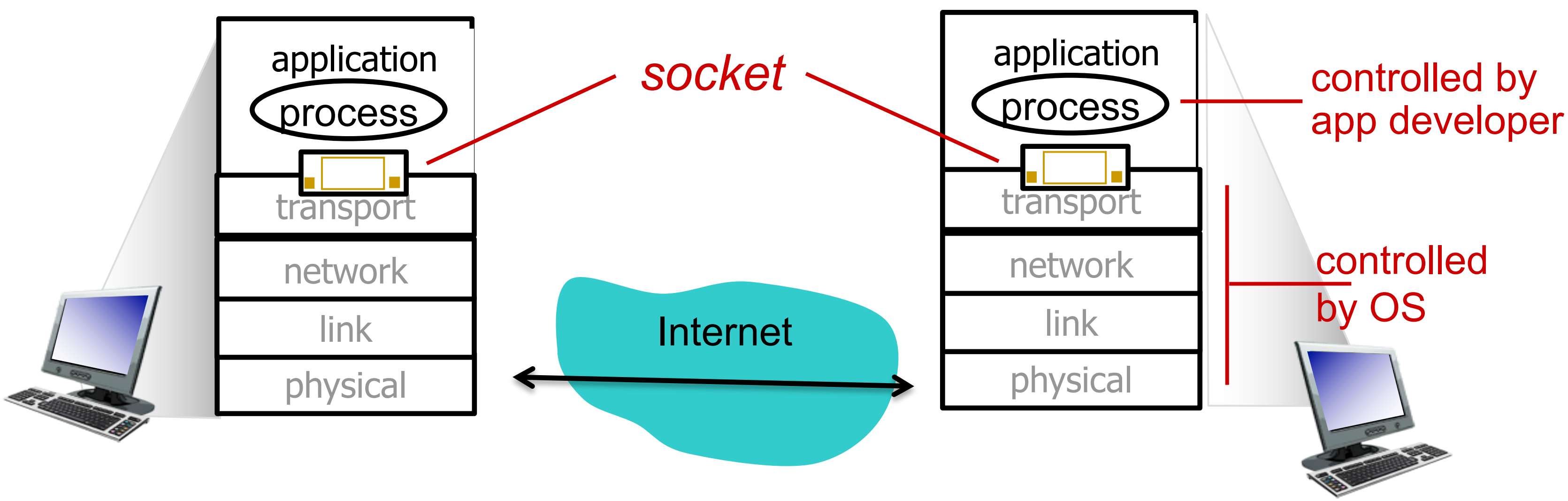
Persistent HTTP:

- 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
- Takes as little as one RTT + file transmission time *total*
  - Assuming connections to server already established
  - Assuming all files requested in parallel

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

Application Example:
1. Client reads a line of characters (data) from its keyboard and sends data 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 the screen

Socket programming *with UDP*

- UDP: no "connection" between client & server
- No handshaking before sending data
- Sender explicitly attaches IP destination address and port # to each packet
- Receiver extracts sender IP address and port# from received packet
- UDP: transmitted data may be lost or received out-of-order
- Application viewpoint:
- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP

Server (running on some IP)

Create socket, port = x:

serverSocket =
socket(AF_INET,SOCK_DGRAM)

Read datagram from
serverSocket

Write reply to
serverSocket
specifying
client address,
port number

Client

Create socket:

clientSocket =
socket(AF_INET,SOCK_DGRAM)

Create datagram with server IP and
port=x; send datagram via
clientSocket

Read datagram from
clientSocket

Close
clientSocket

Example app: UDP client

Python UDPClient

Include Python's socket library

Server IP (Empty=>local system)

IPv4

UDP socket

Create UDP socket for server

Get user keyboard input

Attach server name, port to message; send into socket

Read reply characters from socket into string

Print out received string and close socket

```
from socket import *
serverName = ''
serverPort = 12000

clientSocket = socket(AF_INET,  SOCK_DGRAM)

message = input('Input lowercase sentence: ')

clientSocket.sendto(message.encode(),
                           (serverName, serverPort))

modifiedMessage, serverAddress =
                           clientSocket.recvfrom(2048)
print(modifiedMessage.decode())
clientSocket.close()
```

Example app: UDP server

Python UDPServer

Listening IP (Empty=>local system)

Create UDP socket

Bind socket to local port number 12000

Loop forever

Read from UDP socket into message, getting client's address (client IP and port)

Send upper case string back to this client

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print('The server is ready to receive')
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        clientAddress)
```

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.

Application viewpoint:

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

Client/server socket interaction: TCP

Durham University

server (running on `hostid`)

Create socket,
port=**`x`**, for incoming
request:
serverSocket = socket()
Bind serverSocket to an address

Wait for incoming
connection request
connectionSocket =
serverSocket.accept()

Read request from
connectionSocket

Write reply to
connectionSocket

Close
connectionSocket

TCP
connection setup

client

Create socket,
connect to **`hostid`**, port=**`x`**
clientSocket = socket()
clientSocket.connect()

Send request using
clientSocket

Read reply from
clientSocket

Close
clientSocket

Example app: TCP client

Empty => local machine

Python TCPClient

IP addressing

Use TCP

We will try to connect to remote port 12000 on the local system

Create a new TCP/IP socket

Attach socket to server name, port

Send data into this socket

Receive data from this socket

Close the socket (ending the connection)

```
from socket import *
serverName = ''
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = input('Input lowercase sentence: ')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server: ', modifiedSentence.decode())
clientSocket.close()
```

Example app: TCP server

Python TCPServer

Empty => local machine

Create TCP welcoming socket

Server begins listening for incoming TCP requests

Loop forever

Server waits on accept() for incoming requests, new socket created on return

Read bytes from socket (not *address* as in UDP)

Close connection to this client (but *not* welcoming socket)

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(1)
print('The server is ready to receive')
while True:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024).decode()

    capsSentence = sentence.upper()

    connectionSocket.send(capsSentence.encode())
    connectionSocket.close()
```

Summary

- Application architectures
  - Client-server
  - P2P
- Application service requirements
- Internet transport service model
  - Connection-oriented, reliable: TCP
  - Unreliable, datagrams: UDP

Socket programming:
TCP, UDP sockets

Important themes:

- Control vs. messages
  - in-band, out-of-band
- Centralized vs. decentralized
- Stateless vs. stateful
- Reliable vs. unreliable message transfer
- "Complexity at network edge"