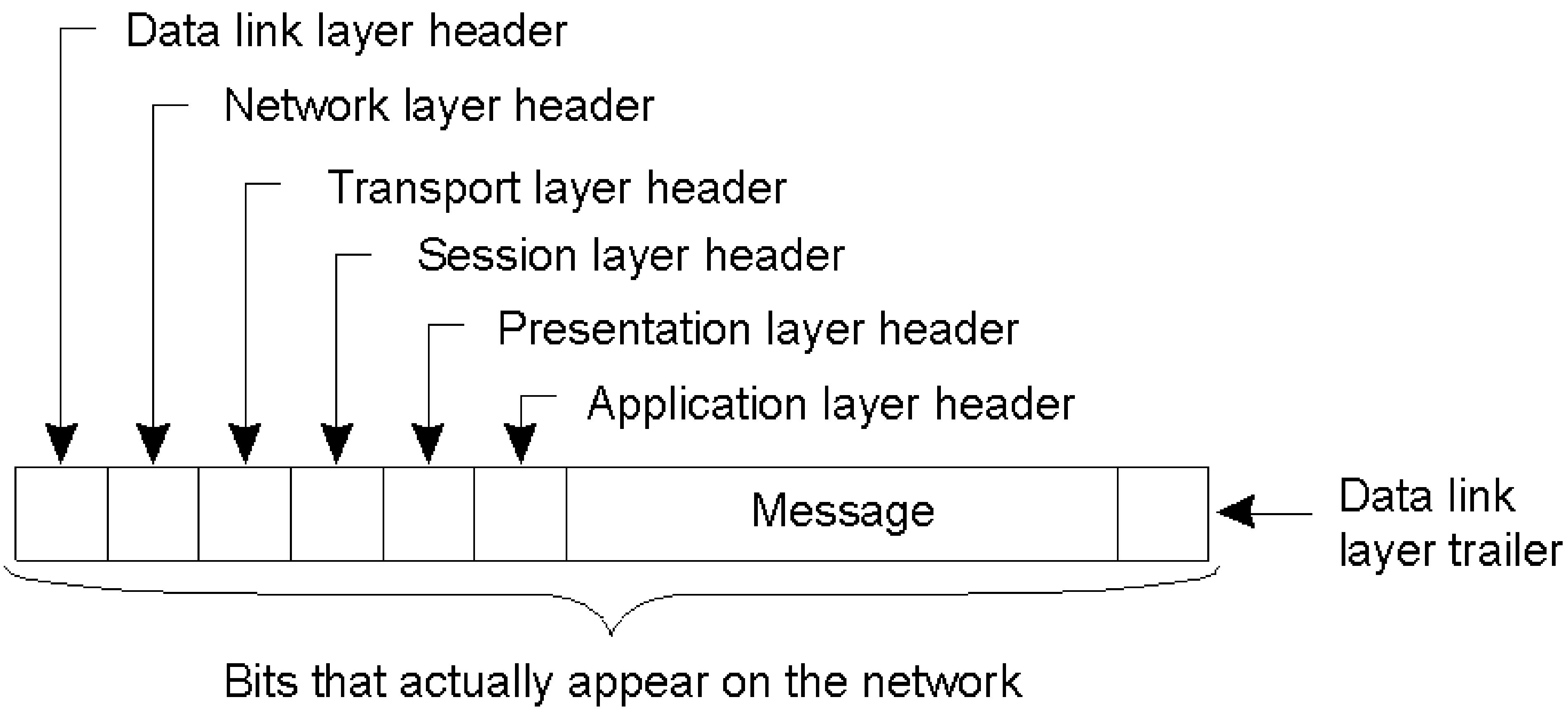
Implementation of UDP and TCP

CS587x Lecture 2
Department of Computer Science
Iowa State University

OSI 7-Layer Model



IP Header

```
 0                   1                   3                   4
 0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  HL   |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- Service
  - Send/receive a packet to/from a remote machine
- Interface 1: **`IP_Send(dest, buf)`**
  - Create a packet (**`IP header + buf`**)
  - Find a routing path to host **`dest`**
  - Send the data in **`buf`** to host **`dest`**
- Interface 2: **`IP_Recv(buf)`**
  - Receive an IP packet
  - Deposit the packet into **`buf`**
  - Return the packet size

- The interface is called by all applications in the same machine
  - How to decide which application gets which packets?
- IP Packets have limited size
  - Each packet can be no more than 64K bytes
- IP is connectionless and does not guarantee packet delivery
  - Packets can be delayed, dropped, reordered, duplicated
- No congestion control

- Each connection links to a specific *port*
  - (srcIP, srcPort, dstIP, dstPort) uniquely identifies each connection
- Totally there are 65535 ports
  - *Well known ports* (0-1023): everyone agrees which services run on these ports
    - e.g., ssh:22, http:80, snmp: 24
    - Access to these ports needs administrator privilege
  - Ephemeral ports (most 1024-65535): given to clients
    - e.g. chatclient gets one of these
    - Port contention rises

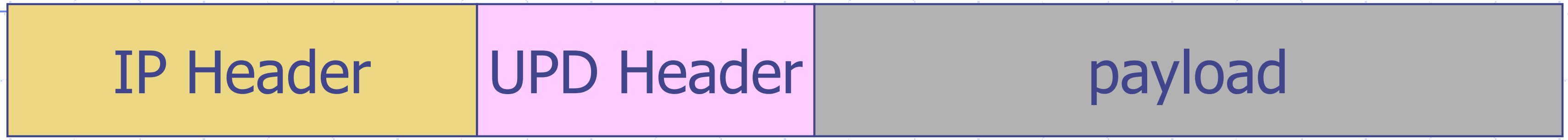
- **Service**
  - Send datagram from (srcIP, srcPort) to (dstIP, dstPort)
  - Service is unreliable, but error detection possible
- **Interface**
  - `UDP_Send(dstIP, buf, port)`
  - `UDP_Recv(buf, port)`
- **Header**

0 — 15	16 — 31
Source Port	Destination Port
UDP length	UDP checksum
Payload (variable)	

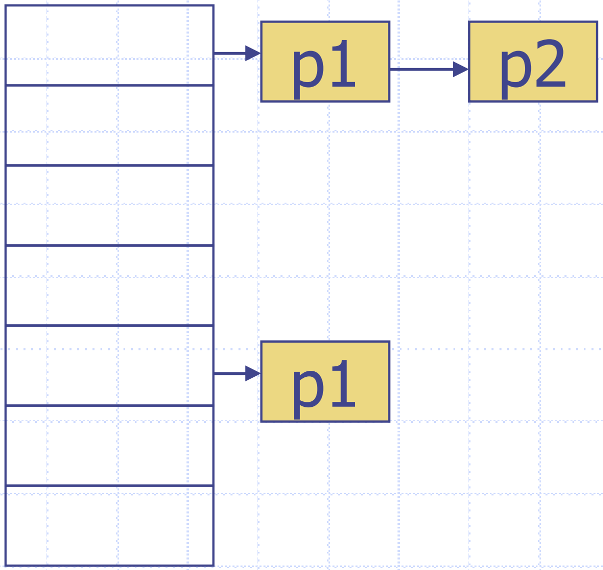
❖ UDP includes UDP header and payload, but not IP header

IP Header	UPD Header	payload

UDP Implementation



Port List



- **`UDP_Recv(buf, port)`**
  1. **`IP_recv(buf)`**
  2. Get **`port`** information from the udp header encoded in **`buf`**
  3. Any listener on **`port`**?
     a) Yes, drop the payload to the message queue of the listener and wake it up (if it is waiting)
     b) No, discard the packet

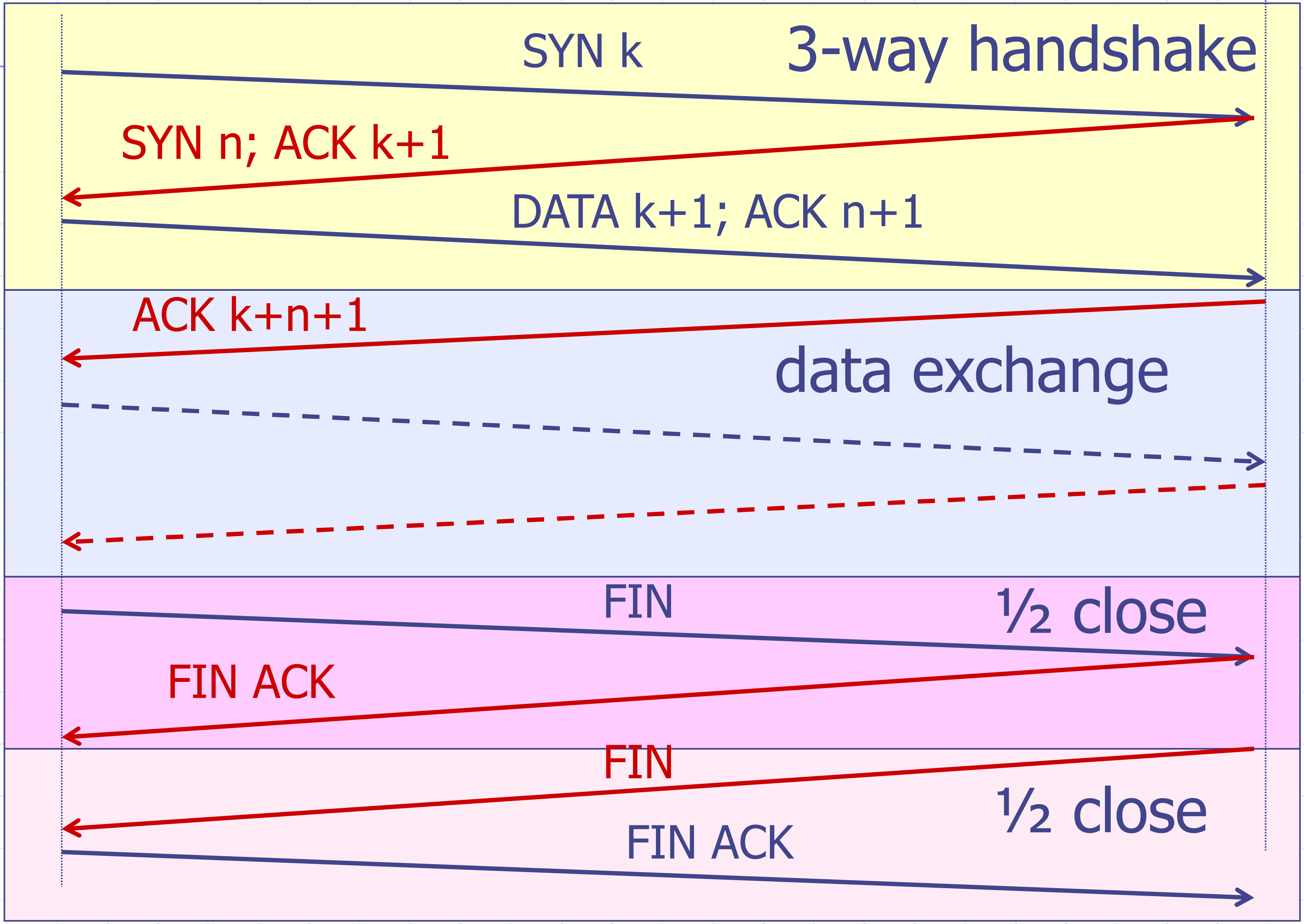
- Provide multiplexing/demultiplexing to IP
- Messages can be of arbitrary length
- Provide reliable and in-order delivery
- Provide congestion control and avoidance

- Start a connection
- Reliable byte stream delivery from (srcIP, srcPort) to (dstIP, dstPort)
- Indication if connection fails: Reset
- Terminate connection

Payload (variable)

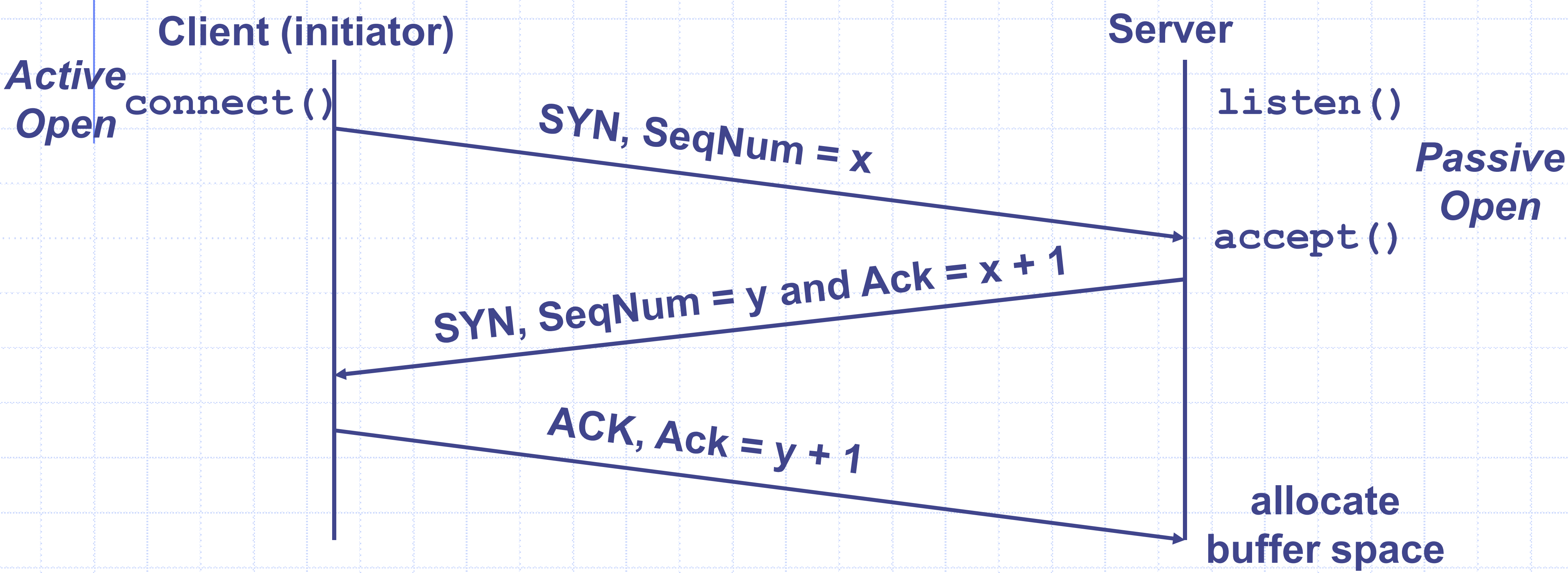
- Sequence number, acknowledgement, and advertised window – used by *sliding-window based flow control*
- Flags:
  - SYN – establishing a TCP connection
  - FIN – terminating a TCP connection
  - ACK – set when Acknowledgement field is valid
  - URG – urgent data; Urgent Pointer says where non-urgent data starts (not defined by standard, but specific to implementation)
  - PUSH – don’t wait to fill segment
  - RESET – abort connection

Connect/Exchange/Terminate



Connection: 3-Way Handshake

- Three messages (i.e., syn, syn, ack) are exchanged before data transmission
- Exchange sequence number, total buffer size and the size of the largest segment that can be handled at each side



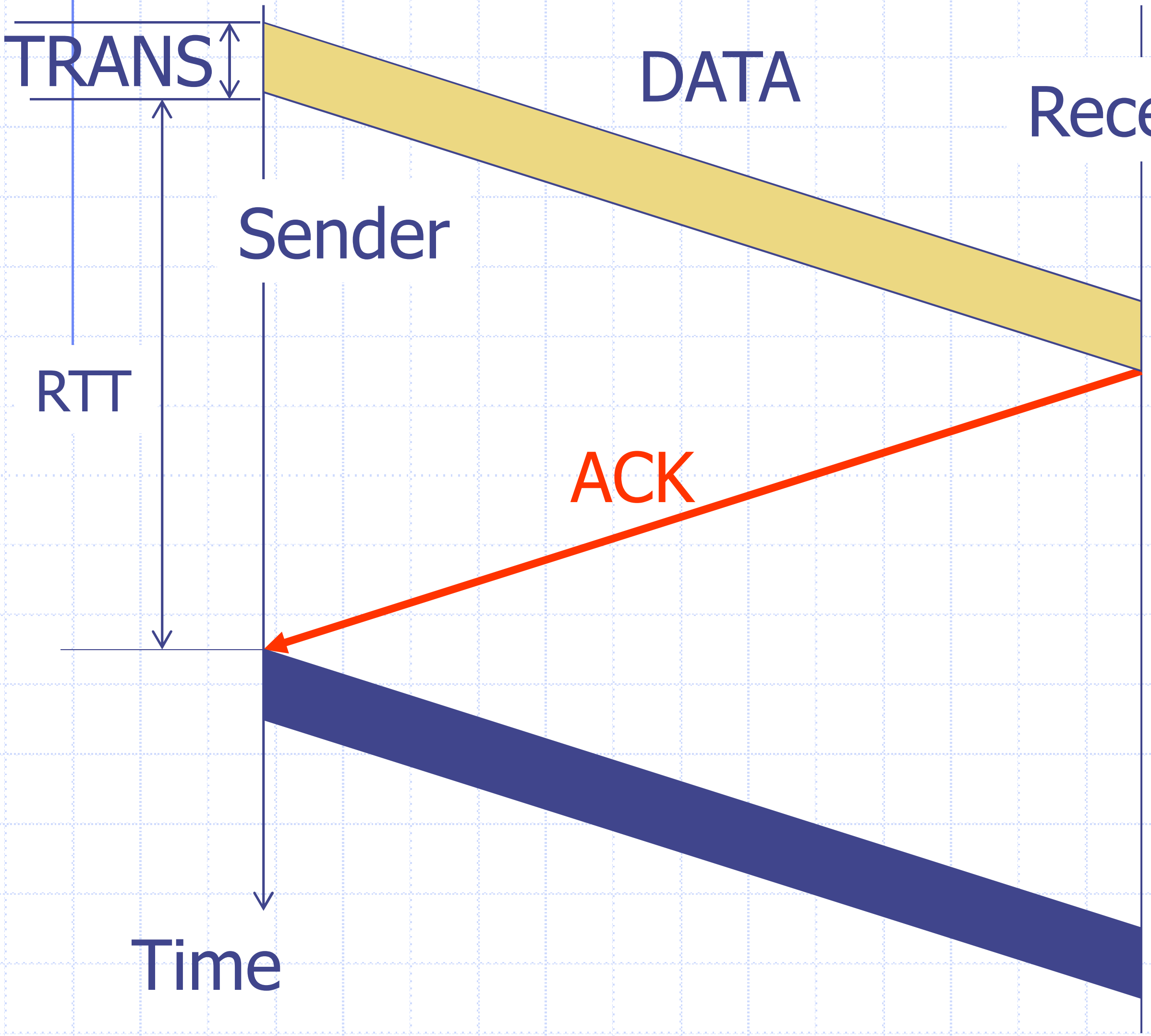
- Three-way handshake adds 1 RTT delay
  - Expensive for small connections such as RPC
- Why?
  - Congestion control: SYN (40 byte) acts as cheap probe
  - Protects against delayed packets from other connection (would confuse receiver)

- How it works: exhausting system resources
  - Using a faked IP address
  - Initiates a TCP connection to a server with a faked IP address
    - Sends a SYN message
    - The server responses with a SYN-ACK
    - Since the address does not exist, the server needs to wait until time out
      - The server never receives the ACK (the final stage of the TCP connection)
  - Repeat with a new faked IP address
    - Repeat at a pace faster than the TCP timeouts release the resources
    - All resources will be in use and no more incoming connection requests can be accepted.
- Some common ways to present
  - Install firewall
    - choose deny instead of reject, which sends a message back to the sender
  - Close all ports that are not in using
  - Deny requests from unusual IP addresses
    - Private address
    - Mulitcast address, etc.

- Four messages (FIN, ACK, FIN, ACK) are exchanged to terminate a connection
  - FIN from B to A
    - B does not transmit any new data, but is still responsible for any corrupted data
  - ACK from A to B
  - FIN from A to B
    - After reading all of the bytes from B, A sends FIN to B
  - ACK from B to A
    - The connection is formally closed

Exchange: Stop & Wait

- Send; wait for ack
- If timeout, retransmit; else repeat

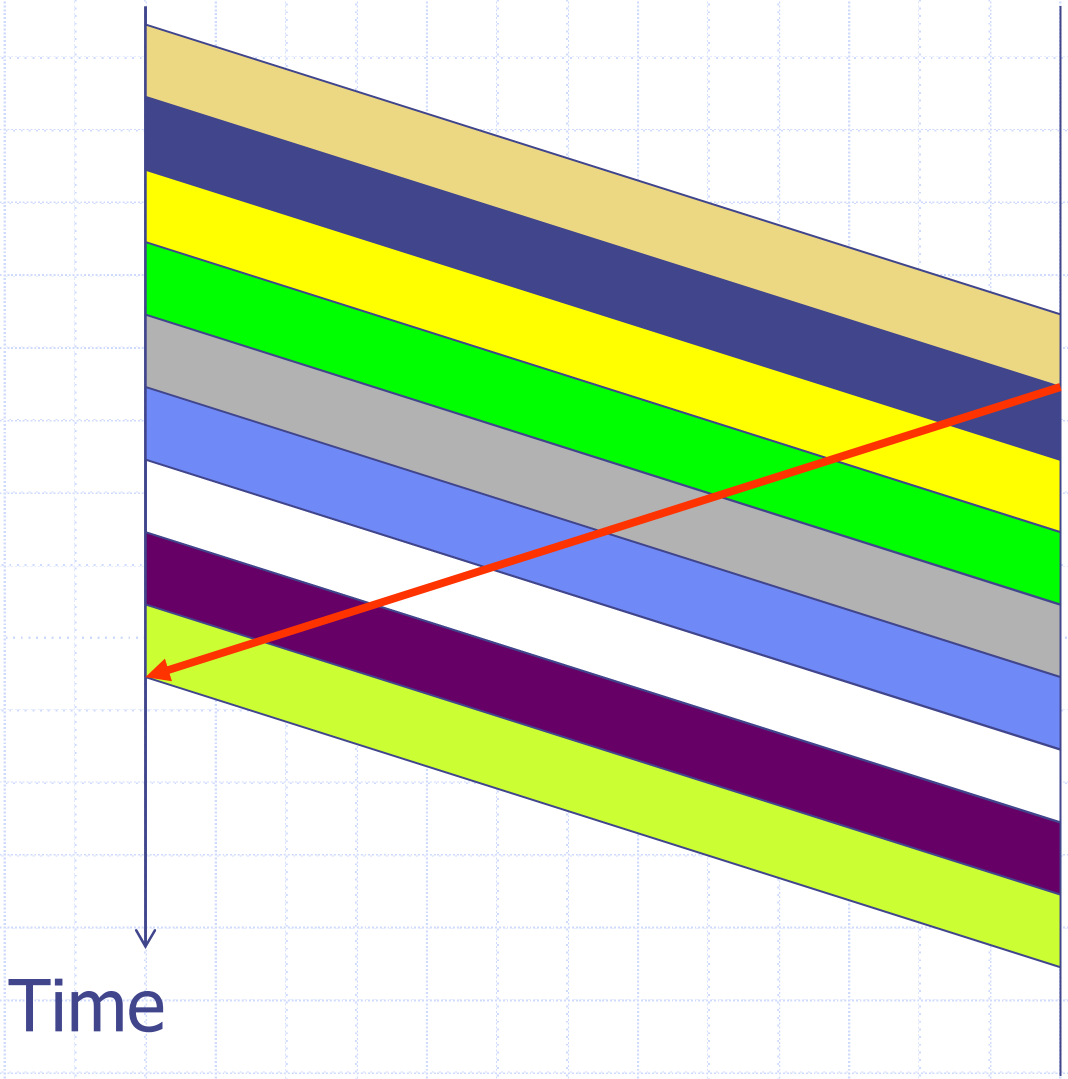


Inefficient if TRANS << RTT

Exchange: Go-Back-n (GBN)

- Sliding Window Protocol
  - Transmit up to n unacknowledged packets/bytes
  - If timeout for ACK(k), retransmit k, k+1, …

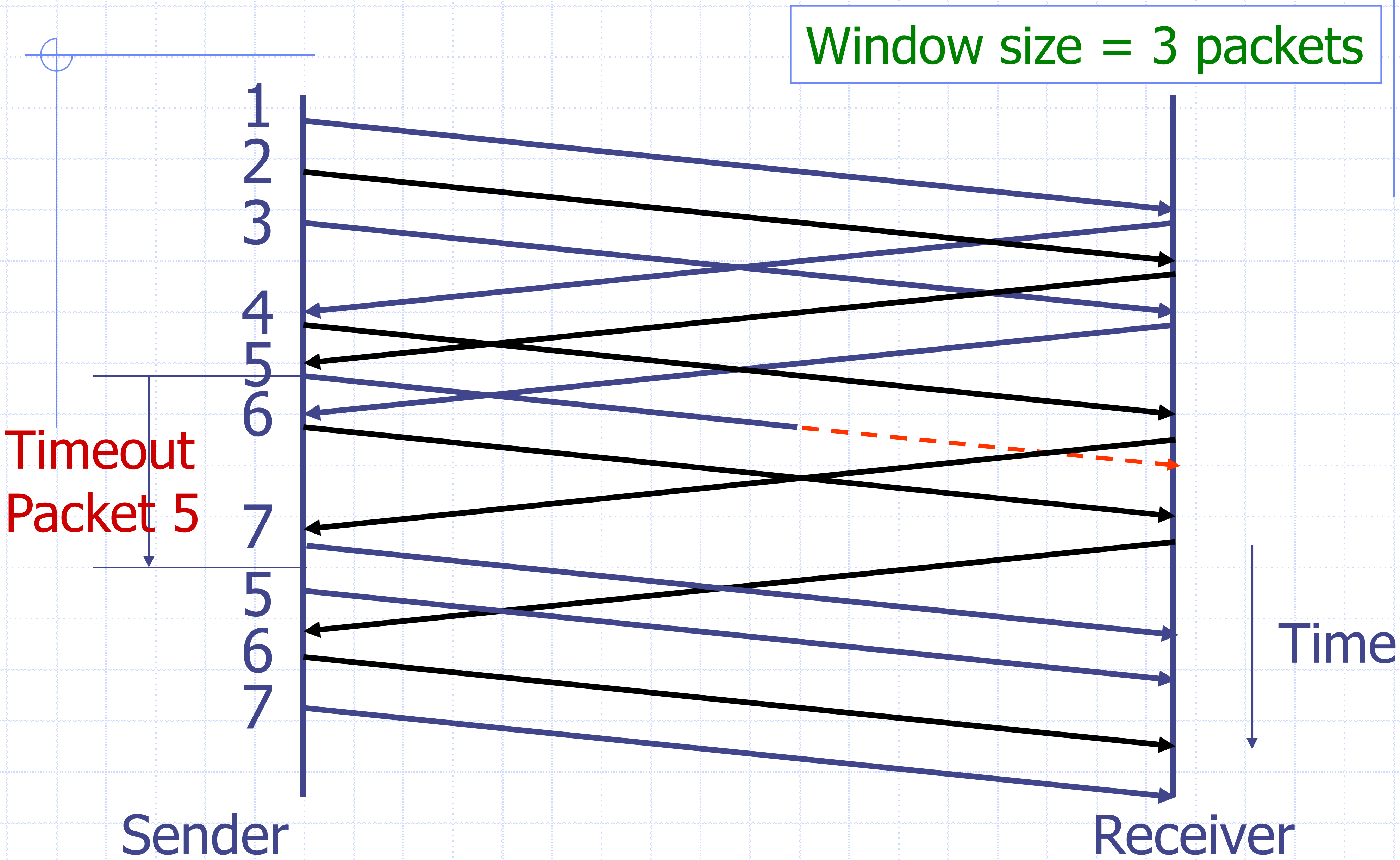
Example without errors



n = 9 packets in one RTT instead of 1

→ Fully efficient

Example with errors

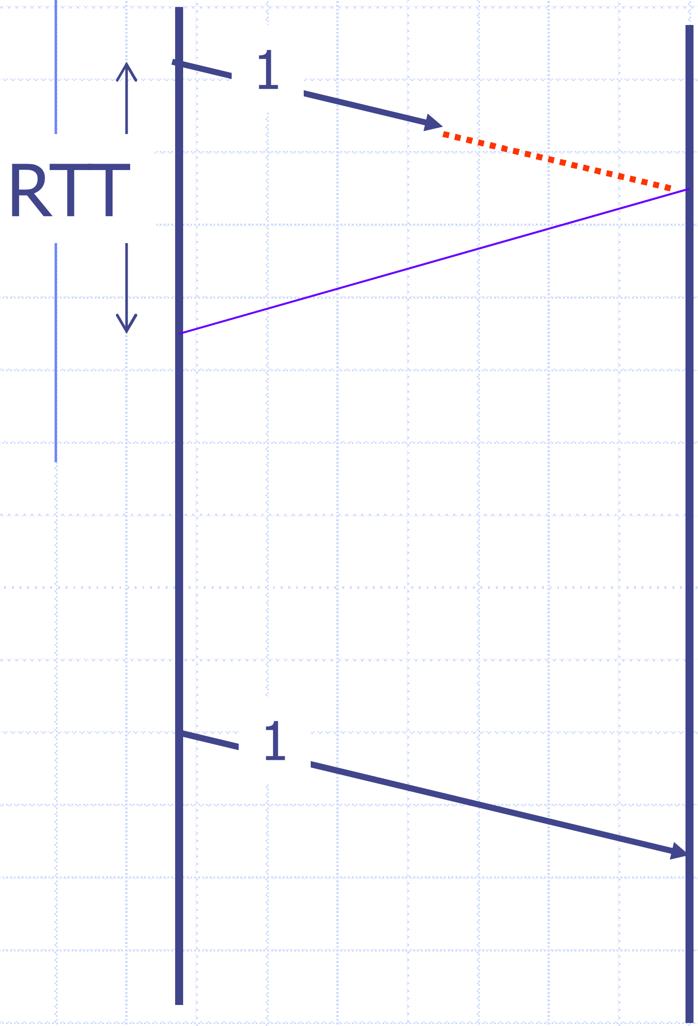


- Pros:
  - It is possible to fully utilize a link, provided the sliding window size is large enough. Throughput is ~ (w/RTT)
  - Stop & Wait is like w = 1.
- Cons:
  - Sender has to buffer all unacknowledged packets, because they may require retransmission
  - Receiver may be able to accept out-of-order packets, but only up to its buffer limits

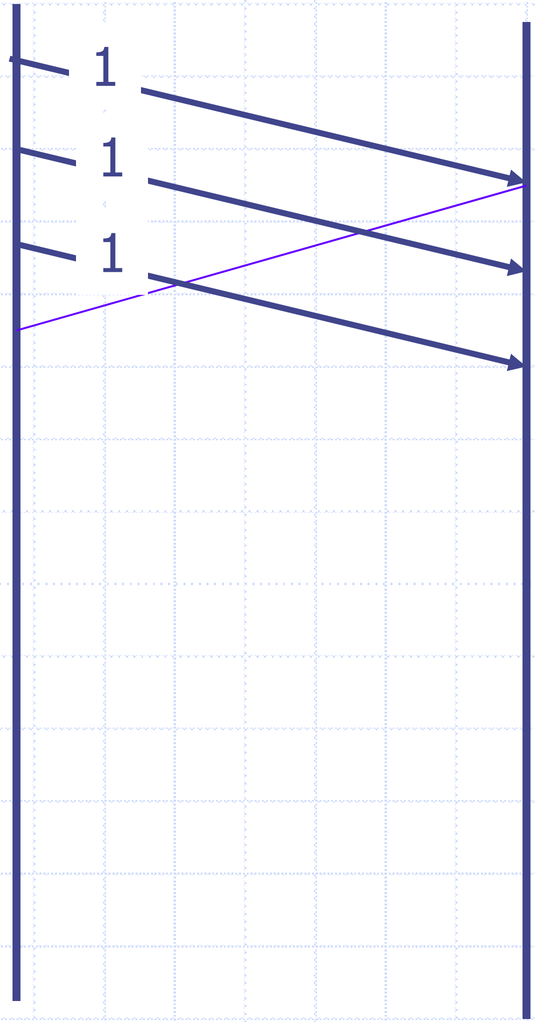
- What size should the window be?
  - Too small:
    - Inefficient, degenerated to S&W when w=1
  - Too large:
    - more buffer required for both sender and receiver
    - Transmitting too fast results in network congestion and packet lost
- Congestion control
  - Slow-start phase
    - Initially set to be 1 or 2
    - Increase the window by 1 for each ACK received (this results in multiplicatively increase)
  - Congestion-avoidance phase
    - The window is increased by only 1 at a time after it is larger than the slow-start threshold (i.e., half of the size that causes congestion)
  - In the case some packet is lost, the window is decreased by half (window / 2).

- The sender needs to set timers in order to know when to retransmit a packet that may have been lost
- How long to set the timer for?
  - Too short: may retransmit before data or ACK has arrived, creating duplicates
  - Too long: if a packet is lost, will take a long time to recover (inefficient)

Illustrations



Timer too long



Timer too short

- The amount of time the sender should wait is about the round-trip time (RTT) between the sender and receiver
  - For link-layer networks (LANs), this value is essentially known
  - For multi-hop WANS, rarely known
- Must work in both environments, so protocol should adapt to the path behavior
- Measure successive ack delays T(n)
  Set timeout = average + 4 deviations

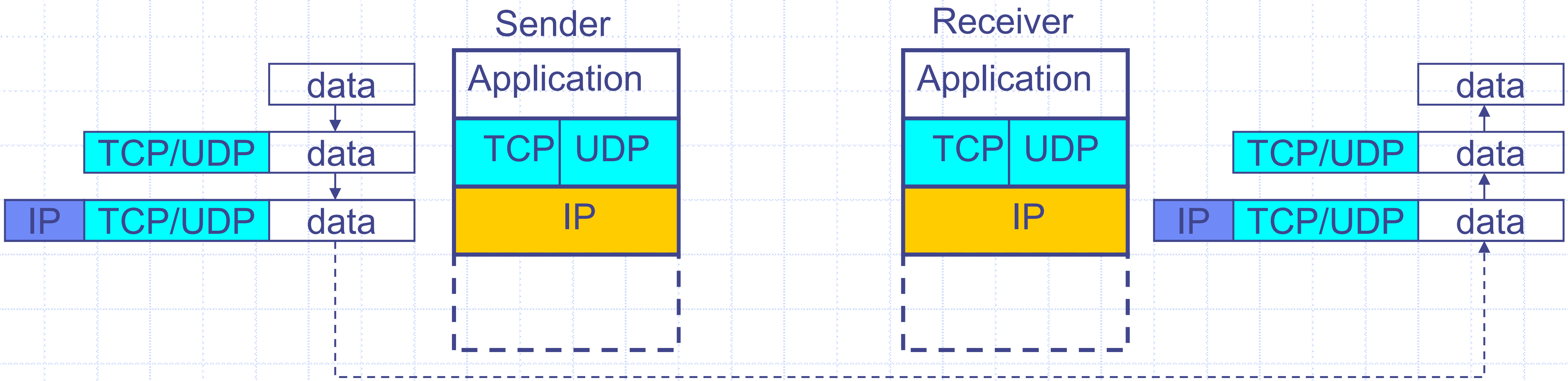
- What exactly should the receiver ACK?
- Some possibilities:
  - ACK every packet, giving its sequence number
  - use *cumulative ACK*, where an ACK for number n implies ACKS for all $k < n$
  - use *negative ACKs* (NACKs), indicating which packet did not arrive
  - use *selective ACKs* (SACKs), indicating those that did arrive, even if not in order

Multi-Source Downloading

- A large file may be available on multiple servers
  - The connections to these servers may not be reliable
- To speed up downloading, a client may download the file from several servers
  - Subject to the limitation of client download bandwidth
- What factors to consider?
  - Which part of the file should be downloaded from a server
  - What happens if some server is down?
  - How about disk I/O cost?

- UDP: Multiplex, detect errors
- TCP: Reliable Byte Stream
  - Connect (3WH); Exchange; Close (4WH)
  - Reliable transmissions: ACKs…
  - S&W not efficient → Go-Back-n
  - What to ACK? (cumulative, …)
  - Timer Value: based on measured RTT

- IP header → used for IP routing, fragmentation, error detection, etc.
- UDP header → used for multiplexing/demultiplexing, error detection
- TCP header → used for multiplexing/demultiplexing, data streaming, flow and congestion control

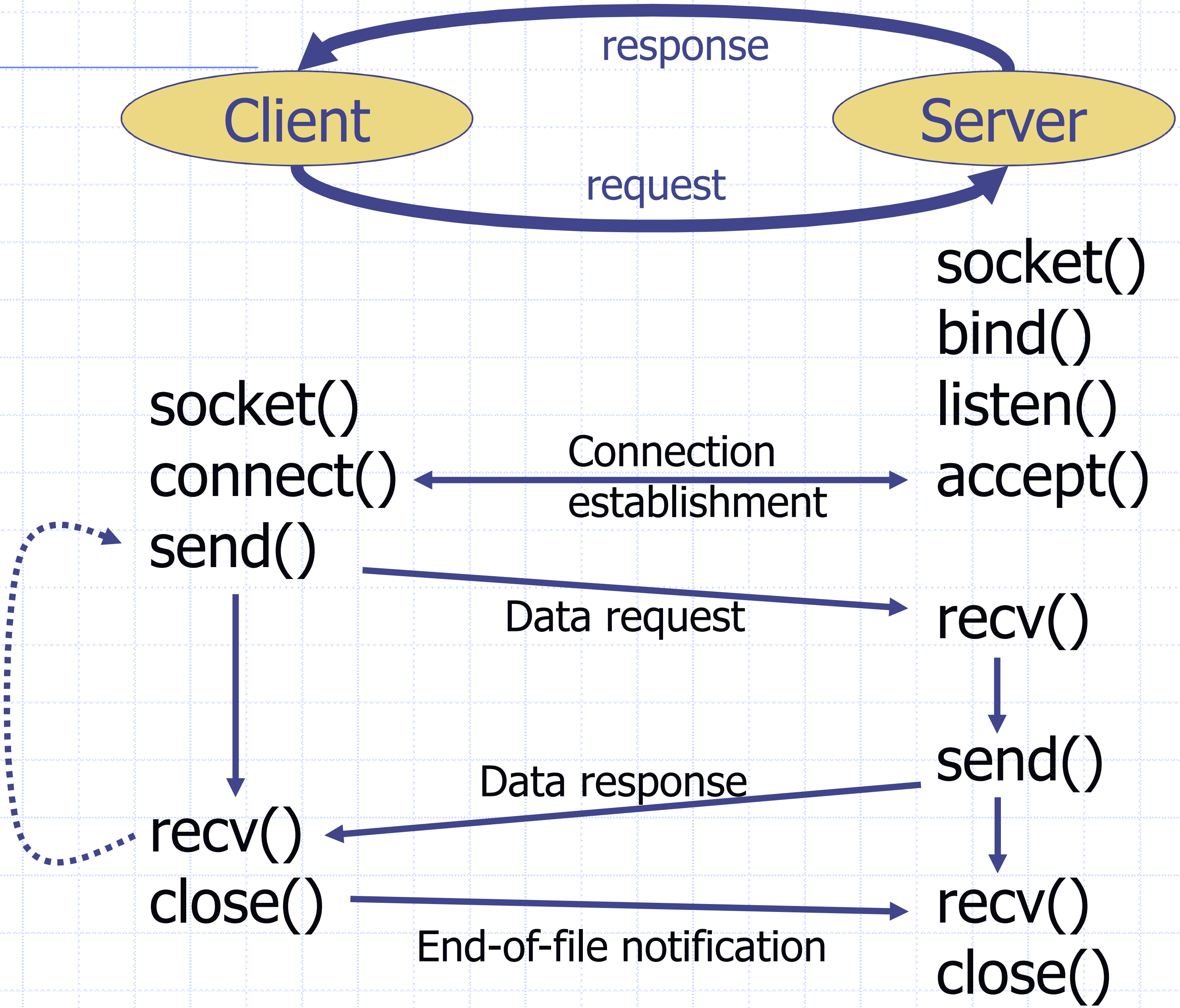


Socket Programming (C/Java)

- Socket provides an interface for a programmer to write applications that communicate between two hosts across IP network
- Socket types of interest
  - SOCK_STREAM
    - Maps to TCP in the AF_INET family
  - SOCK_DGRAM
    - Maps to UDP in the AF_INET family

- Client requests service from server
- Server responds with sending service or error message to client

Simple Client-Server Example



- Create stream socket (*socket()*)
- Connect to server (*connect()*)
- While still connected:
  - send message to server (*send()*)
  - receive (*recv()*) data from server and process it

Getting the file descriptor

```
int cSock;
if ((cSock = socket(AF_INET, SOCK_STREAM, NULL)) < 0)
{
    perror("socket");
    printf("Failed to create socket\n");
    abort ();
}
```

Connecting to Server

```
struct hostent *host = gethostbyname(argv[1]);
unsigned int svrAddr = *(unsigned long *) host->h_addr_list[0];
unsigned short svrPort = atoi(argv[2]);

struct sockaddr_in sin;
memset (&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = svrAddr;
sin.sin_port = htons(svrPort);

if (connect(cSock, (struct sockaddr *) &sin, sizeof(sin)) < 0)
{
    fprintf(stderr, "Cannot connect to server\n");
    abort();
}
```

```
int send_packets(char *buffer, int buffer_len)
{
    sent_bytes = send(cSock, buffer, buffer_len, 0);
    if (send_bytes < 0)
    {
        fprintf(stderr, "cannot send. \n");
    }
    return 0;
}
```

- Needs socket descriptor,
- Buffer containing the message, and
- Length of the message

Receiving Packets

```
int receive_packets(char *buffer, int bytes)
{
    int received = 0;
    int total = 0;
    while (bytes != 0)
    {
        received = recv(cSock, buffer[total], bytes);
        if (received == -1) return -1;
        if (received == 0) return total;
        bytes = bytes - received;
        total = total + received;
    }
    return total;
}
```

- create stream socket (*socket()*)
- Bind port to socket (*bind()*)
- Listen for new client (*listen()*)
- user connects (*accept()*)
- data arrives from client (*recv()*)
- data has to be send to client (*send()*)

- Server application needs to call bind() to tell operating system (i.e. network layer) which port to listen
- Client application does not need bind()
  - Any port can be used to send data
  - The server application will get the port number of the client application through the UDP/TCP packet header
- Server port must be known by client application in order to connect to the server
- How to handle if a port has been used by another application?

Server Programming

```
struct hostent *host = gethostbyname (argv[1]);
unsigned int svrAddr = *(unsigned long *) host->h_addr_list[0];
unsigned short svrPort = atoi (argv[2]);

struct sockaddr_in sin;
memset (&sin, 0, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = svrAddr;
sin.sin_port = htons (svrPort); /* network byte order (big-endian) */

int svrSock = socket( AF_INET, SOCK_STREAM, 0 );
if (bind(svrSock, (struct sockaddr *) &sin, sizeof(sin)) < 0)
{
    fprintf(stderr, "Cannot bind to network\n");
    abort();
}
listen(svrSock, 5); /* maximum 5 connections will be queued */
while (1)
{
  int cltSock = accept(svrSock, (struct sockaddr *)&cli_addr, &clilen );
/* launch a new thread to take care of this client connection */
/* cli_addr contains the address of the connecting client */
/* clilent is the buffer length that is valid in cli_addr */
/* both cli_addr and clileng are optional */
}
```

Java Socket Programming

- TCP stream
  - java.net.Socket
  - java.net.ServerSocket
- UDP packet
  - Java.net.DatagramPacket
  - java.net.DatagramSocket

- java.net.Socket is used by clients to make a bi-directional connection with server
- Socket constructors
  - Socket(String hostname, int port)
  - Socket(InetAddress addr, int port)
  - Socket(String hostname, int port, InetAddress localAddr, int localPort)
    /* specify a specific NIC and port to use */
  - Socket(InetAddress addr, int port, InetAddress localAddr, int localPort)
- Creating socket

  Socket csweb = new Socket(“www.cs.iastate.edu", 80);

```
try
{
    String s;
    Socket socket = new Socket("www.cs.iastate.edu", 80);
    BufferedReader reader = new BufferedReader(
                 new InputStreamReader(socket.getInputStream()));
    PrintStream pstream = new PrintStream(socket.getOutputStream());
    pstream.println("GET /");
    while ((s = reader.readLine()) != null)
    {
       System.out.println(s);
    }
}
catch (Exception e)
{
    System.err.println("Error: " + e);
}
```

- Socket() attempts to connect the server immediately
- Cannot set or change remote host and port
- Socket constructors may block while waiting for the remote host to respond

- void setReceiveBufferSize()
- void setSendBufferSize()
- void setTcpNoDelay()
- void setSoTimeout()

- ServerSocket is used by server to accept client connections
- ServerSocket constructor

```
public ServerSocket(int port)
public ServerSocket(int port, int backlog)
public ServerSocket(int port, int backlog, InetAddress networkInterface)
```

- Creating a ServerSocket

```
ServerSocket ss = new ServerSocket(80, 50);
```

- A closed ServerSocket cannot be reopened

A Simple Server

```
try
{
    ServerSocket ss = new ServerSocket(2345);
    Socket s = ss.accept();
    PrintWriter pw = new PrintWriter(s.getOutputStream());
    pw.println("Hello There!");
    pw.println("Goodbye now.");
    s.close();
}
catch (IOException ex)
{
    System.err.println(ex);
}
```

Sending UDP Datagrams

1. Convert the data into byte array.
2. Create a DatagramPacket using the array
3. Create a DatagramSocket using the packet and then call send() method

Example

```
InetAddress dst = new InetAddess(“cs.iastate.edu");
String s = “This is my datagram packet"
byte[] b = s.getBytes();
DatagramPacket dp = new DatagramPacket(b, b.length, dst, 2345);
DatagramSocket sender = new DatagramSocket();
sender.send(dp);
```

Note: DatagramPacket object can be reused (e.g., setting different dst and port).

Receiving UDP Datagrams

1. Construct an empty DatagramPacket (with a buffer)
2. Pass the object to a DatagramSocket (with a port)
3. Call the DatagramSocket's receive() method
4. The calling thread blocks until a datagram is received

```
byte buffer = new byte[1024];
DatagramPacket incoming = new DatagramPacket(buffer, buffer.length);
DatagramSocket ds = new DatagramSocket(2345);
ds.receive(incoming);
byte[] data = incoming.getData();
String s = new String(data, 0, incoming.getLength());
System.out.println("Port" +  incoming.getPort()  +
                   " on "  +  incoming.getAddress()  +
                   " sent this message:");
System.out.println(s);
```

A Mistake You Want to Avoid

```
byte[] buf = new byte[1024];
DatagramPacket incoming = new DatagramPacket(buf, buf.length);
DatagramSocket ds = new DatagramSocket(2345);
for (;;)
{
    ds.receive(incoming);
    byte[] data = incoming.getData();
   new DataProcessor(data).start();
}

class DataProcessor(byte[] data) extends Thread
{
    // processing data[] …
}
```

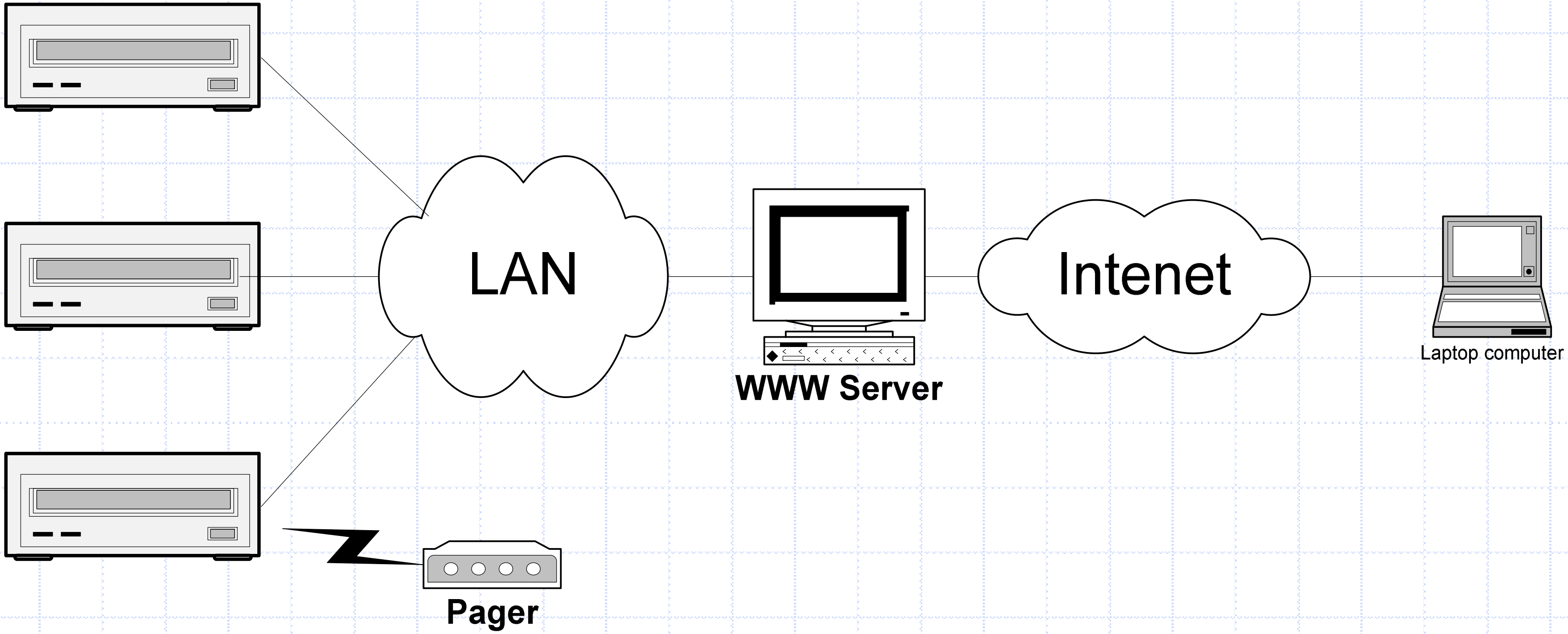
Correct Way

```
byte[] buf = new byte[1024];
DatagramPacket incoming = new DatagramPacket(buf, buf.length);
DatagramSocket ds = new DatagramSocket(2345);
for (;;)
{
    ds.receive(incoming);
    byte[] data = new byte[incoming.getLength()];
    System.arraycopy(incoming.getData(), 0, data, 0, data.length);
    new DataProcessor(data).start();
}

class DataProcessor(byte[] data) extends Thread
{
    // processing data[] …
}
```

- RAID Management
  - Monitor the health condition of RAID subsystems
    - disks, fans, power supplies, temperatures, etc.
  - Report any failure instantly
  - Provide disaster recovery
    - array rebuild, spare disks reassign, etc.

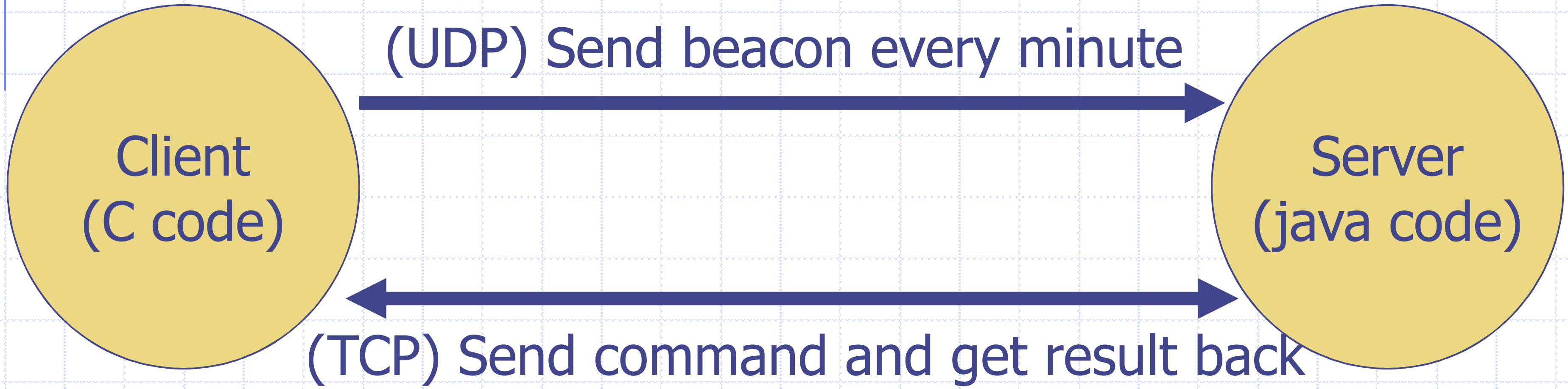
Remote and Centralized Storage Management



Homework #1

Assigned on Sept. 7, 2005

Due on 3:00PM, Sept. 21, 2005



- BeaconSender: Send the following message to server every minute using UDP datagram

```
struct BEACON
{
      int    ID;             // randomly generated during startup
      int    StartUpTime;    // the time when the client starts
      char   IP[4];          // the IP address of this client
      int    CmdPort;        //  the client listens to this port for cmd
}
```

- CmdAgent: Receive and execute remote commands and send results back using TCP socket. You implement two commands:

```
(1) void GetLocalOS(char OS[16], int *valid)
    // OS[16] contains the local operation system name
    // valid = 1 indicates OS is valid
(2) void GetLocalTime(int *time, int *valid)
    // time contains the current system clock
    // valid = 1 indicates time is valid
```

- **BeaconListener thread**
  - Receive beacons sent by clients
  - For each new client, spawn a thread called ClientAgent
- **ClientAgent(beacon) thread**
  - Send command GetLocalOS() to the corresponding client
  - Get the result back and display the OS
  - Send command GetLocalTime() to the corresponding client
  - Get the result back and display the execution time