Networks Lecture 7

Paolo Ciancarini

Innopolis University

February 7, 2022

Source of the material

- This lecture is based on the following resources
  - Chapter 3 of Computer Networking: A Top Down Approach (8th edition) by Jim Kurose and Keith Ross
  - Based on slides of D. Choffnes Northeastern Unies
  - The material is aligned and add/deleted according to the need of the students.

Topic of the lecture

- Connection-oriented transport: TCP
  - Connection management
  - Segment structure
  - Reliable data transfer
  - Flow control
- Principles of congestion control
- TCP congestion control

Topic of the tutorial

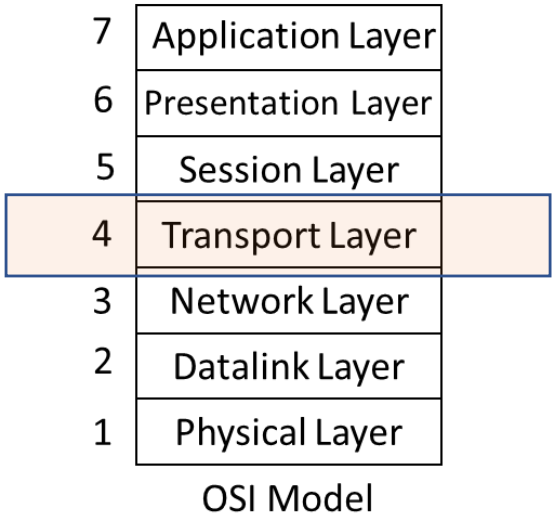
- Packet capturing using tcpdump/wireshark

Topic of the lab

- Packet capturing via TCP/IP

Recap

- So far we discussed about Transport Layer Services
  - TCP Protocol
  - UDP Protocol
- Multiplexing and Demultiplexing
- TCP Error Control
  - Checksum
  - Acknowledgment
  - Time-out



Summary of reliable data transfer mechanisms and their uses

Mechanism	Use, comments
Checksum	Used to detect bit errors in a transmitted packet
Timer	Used to timeout/retransmit a packet, possibly because the packet (or its ACK) was lost within the channel. Because timeouts can occur when a packet is delayed but not lost (premature timeout), or when a packet has been received by the receiver but the receiver-to-sender ACK has been lost, duplicate copies of a packet may be received by a receiver.
Sequence number	Used for sequential numbering of packets of data flowing from sender to receiver. Gaps in the sequence numbers of received packets allow the receiver to detect a lost packet. Packets with duplicate sequence numbers allow the receiver to detect duplicate copies of a packet
Acknowledgment	Used by the receiver to tell the sender that a packet or set of packets has been received correctly. Acknowledgments will typically carry the sequence number of the packet or packets being acknowledged. Acknowledgments may be individual or cumulative, depending on the protocol.
Negative acknowledgment	Used by the receiver to tell the sender that a packet has not been received correctly. Negative acknowledgments will typically carry the sequence number of the packet that was not received correctly
Window, pipelining	The sender may be restricted to sending only packets with sequence numbers that fall within a given range. By allowing multiple packets to be transmitted but not yet acknowledged, sender utilization can be increased over a stop-and-wait mode of operation. We'll see shortly that the window size may be set on the basis of the receiver's ability to receive and buffer messages, or the level of congestion in the network, or both.

TCP: Overview RFCs: 793,1122,1323, 2018, 2581

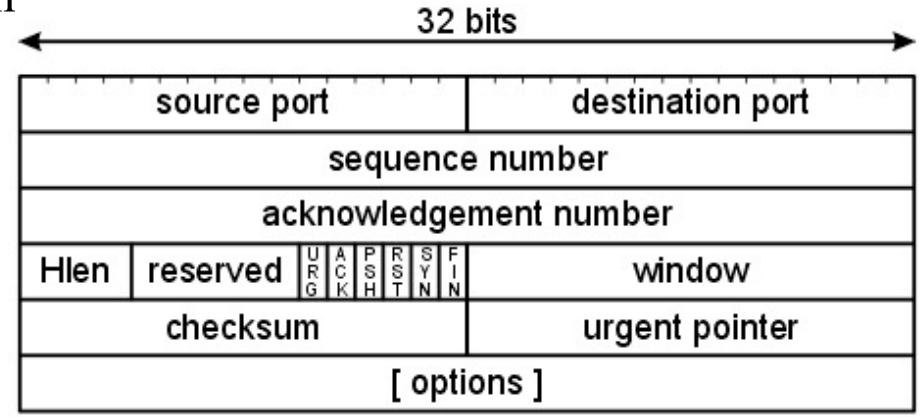
- Point-to-point:
  - one sender, one receiver
- Reliable, in-order *byte stream:*
  - no "message boundaries"
- Pipelined:
  - TCP congestion and flow control set window size
- Full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size

Today our focus!!

- Connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- Flow controlled:
  - sender will not overwhelm receiver

Connection Setup

- Why do we need connection setup?
  - To establish state on both hosts
  - Most important state: sequence numbers
    - Count the number of bytes that have been sent
    - Initial value chosen at random (It is a strongly random number: there are security problems if anybody on the internet can guess the sequence number, as they can easily forge packets to inject into the TCP stream)
- Important TCP flags (1 bit each)
  - SYN – synchronization, used for connection setup
  - ACK – acknowledge received data
  - FIN – finish, used to tear down connection

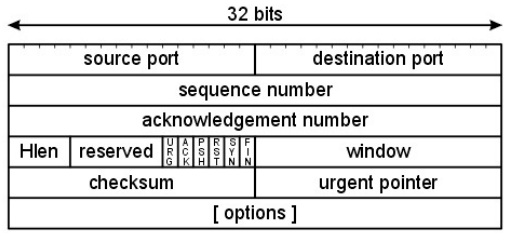


Connection Setup – Code Example

- The client establishes a connection with the server by calling **`connect()`**

```
int status = connect(sockid, &foreignAddr, addrlen);
```

  - **`sockid`**: integer, socket to be used in connection
  - **`foreignAddr`**: struct sockaddr: address of the passive participant
  - **`addrlen`**: integer, sizeof(name)
  - **`status`**: 0 if successful connect, -1 otherwise



Connection Setup – Code Example

```
//try to connect with server
int SocketConnect(int hSocket) {
    int iRetval=-1;
    int ServerPort = 90190;
    struct sockaddr_in remote={0};

    remote.sin_addr.s_addr = inet_addr("127.0.0.1"); //Local Host
    remote.sin_family = AF_INET;
    remote.sin_port = htons(ServerPort);

    iRetval = connect(hSocket , (struct sockaddr *)&remote , sizeof(struct
                                                        sockaddr_in));

    return iRetval;
}
```

```
//accept connection from an incoming client
sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t*)&clientLen);
if (sock < 0) {
    perror("accept failed");
    return 1;}
printf("Connection accepted\n");
```

Connection Setup – Code Example

```
//sockaddr
struct sockaddr {
     sa_family_t sa_family;
     char sa_data[14];
}

struct sockaddr_in {
     sa_family_t sin_family;    /* address family: AF_INET */
     in_port_t sin_port;        /* port in network byte order */
     struct in_addr sin_addr;   /* internet address */
};

struct in_addr {
     uint32_t s_addr; /* address in network byte order */
};

struct sockaddr_in saddr;
     int sockfd;
     unsigned short port = 80;
     sockfd=socket(AF_INET, SOCK_STREAM, 0);
     memset(&saddr, '\0', sizeof(saddr));          // zero structure out
     saddr.sin_family = AF_INET;                   // match the socket() call
     saddr.sin_addr.s_addr = htonl(INADDR_ANY);    // bind to any local address
     saddr.sin_port = htons(port);                 // specify port to listen on
     if((bind(sockfd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0) { // bind!
     printf("Error binding\n");
     ...
     }
```

Connection Setup – Code Example

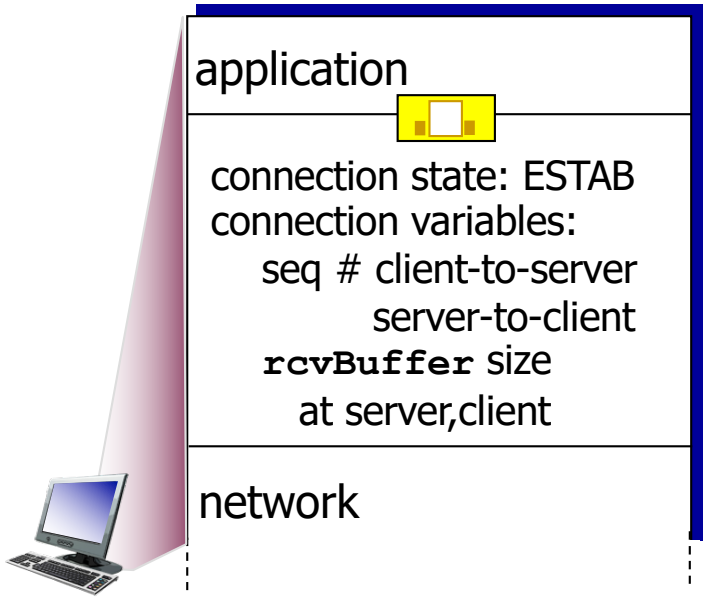
```
/* * Address families. */
#define AF_UNSPEC 0 /* unspecified */
#define AF_LOCAL 1 /* local to host (pipes, portals) */
#define AF_UNIX AF_LOCAL /* backward compatibility */
#define AF_INET 2 /* internetwork: UDP, TCP, etc. */
#define AF_IMPLINK 3 /* arpanet imp addresses */
#define AF_PUP 4 /* pup protocols: e.g. BSP */
#define AF_CHAOS 5 /* mit CHAOS protocols */
#define AF_NS 6 /* XEROX NS protocols */
#define AF_ISO 7 /* ISO protocols */
#define AF_OSI AF_ISO
#define AF_ECMA 8 /* European computer manufacturers */
#define AF_DATAKIT 9 /* datakit protocols */
#define AF_CCITT 10 /* CCITT protocols, X.25 etc */
#define AF_SNA 11 /* IBM SNA */
#define AF_DECnet 12 /* DECnet */
#define AF_DLI 13 /* DEC Direct data link interface */
#define AF_LAT 14 /* LAT */
#define AF_HYLINK 15 /* NSC Hyperchannel */
```

An address family defines a specific addressing format. Applications that use the same addressing family have a common scheme for addressing socket endpoints.

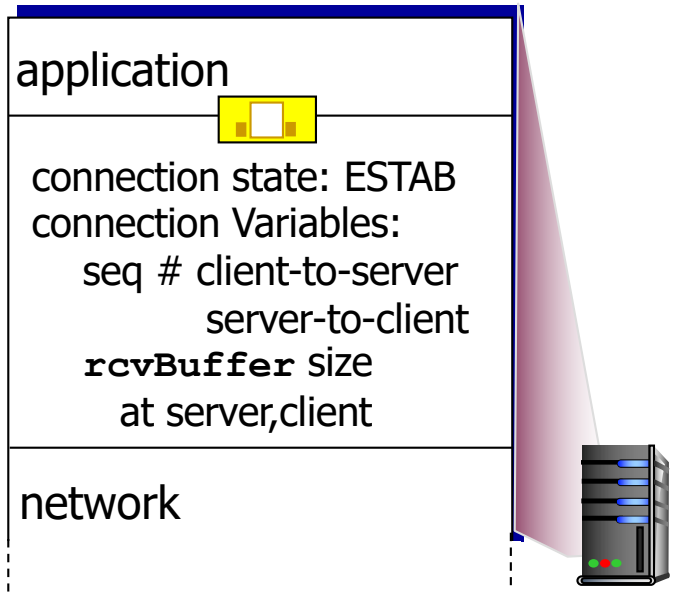
Connection Management

Before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters




```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

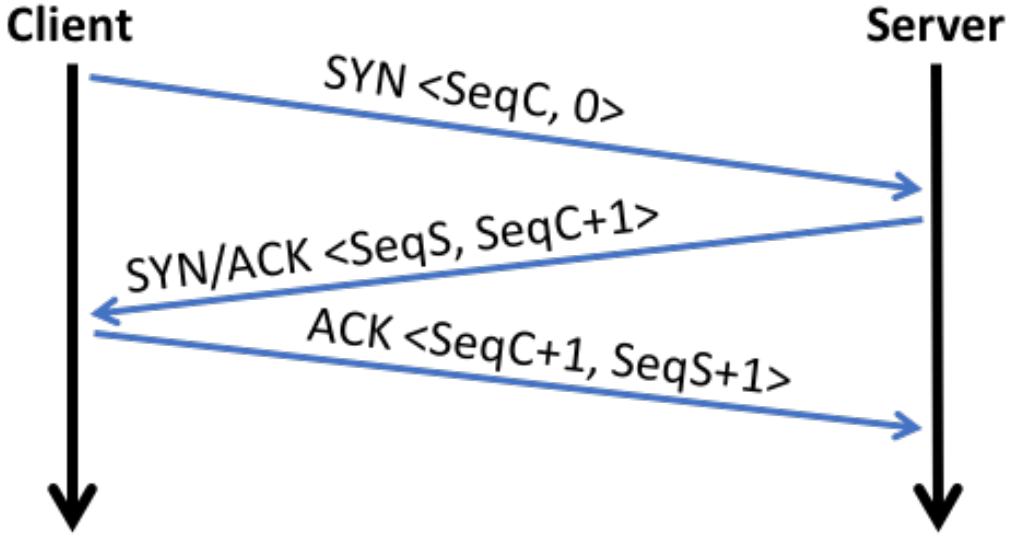



```
Socket connectionSocket =
   welcomeSocket.accept();
```

Connection Management – Code Example

```
//Create socket
hSocket = SocketCreate();
if(hSocket == -1)
{
  printf("Could not create socket\n");
  return 1;
}

printf("Socket is created\n");

//Connect to remote server
if (SocketConnect(hSocket) < 0)
{
  perror("connect failed.\n");
  return 1;
}

printf("Successfully connected with server\n");
```

Three Way Handshake



- Each side:
  - Notifies the other of starting sequence number
  - ACKs the other side's starting sequence number

- The three-way handshake is necessary because both parties need to **syn**chronize their segment sequence numbers used during their transmission.

- For this, each of them sends (in turn) a SYN segment with a sequence number set to a random value *n*, which then is **ack**nowledged by the other party via a ACK segment with a sequence number set to *n+1*.

Why 3-ways handshake? Not 2-ways

- The 3-ways handshake is necessary because both parties need to synchronize their segment sequence numbers used during their transmission.

- So, they(in turn) send a SYN segment with a sequence number set to a value n, which then is acknowledged by the other party via a ACK segment with a sequence number set to n+1.

- Suppose that client does not send ACK(as in 2-ways handshake). Now there might exist a case where seq number of client is not synchronized, but the server will assume that it is synchronized. This could cause a problem.

Connection Setup Issues

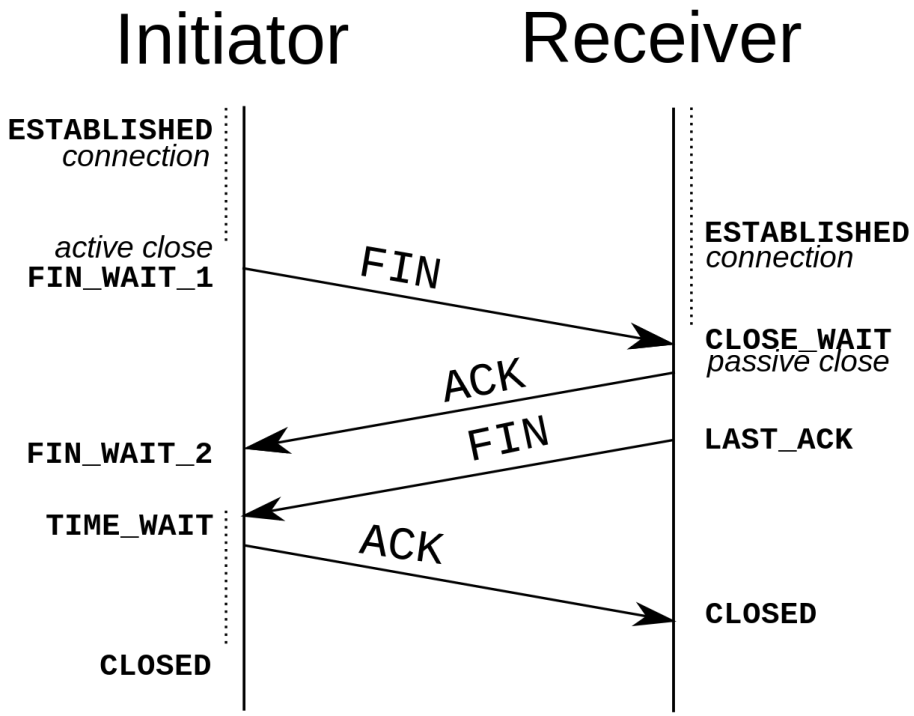
- Connection confusion
  - How to disambiguate connections from the same host?
  - Random sequence numbers
- Source spoofing
  - Kevin Mitnick
  - Need good random number generators!
- Connection state management
  - Each SYN allocates state on the server
  - SYN flood = denial of service attack
  - Solution: SYN cookies

- What is Spoofing?
  - The word '**spoof**' means to trick or deceive
  - A **spoofing attack** is when a malicious party impersonates another device or user on a network in order to launch attacks against network hosts, steal data, spread malware or bypass access controls.

- Spoofing + sequence prediction to hijack connections
- SYN cookie: special sequence number sent in SYNACK so that when ACK comes back SYN cookie value can be reconstructed

TCP: Closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled



Connection Close: C code

```
status = close(sockid);
```

- sockid: the file descriptor (socket being closed)
- status: 0 if successful, -1 if error

- Closing a socket
  - closes a connection (for stream socket)
  - frees up the port used by the socket

Topic of the lecture

- Connection-oriented transport: TCP
  - Connection management
  - Segment structure
  - Reliable data transfer
  - Flow control
- Principles of congestion control
- TCP congestion control

TCP Segment & IP Fragment

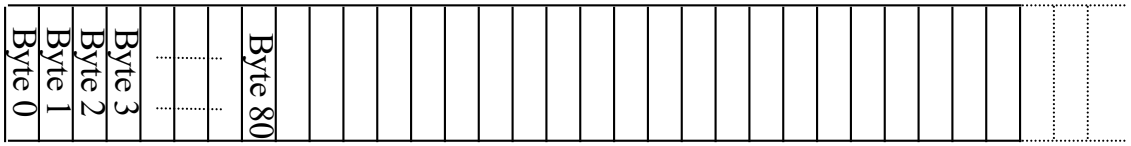
- In **TCP/IP**, a packet is called as a **'Segment'** at Transport Layer and as a **“Fragment”** at Network Layer.

- When Maximum Transmissible Unit(MTU) is greater than some threshold then Sender will do the **Segmentation** (splitting the **payload** into smaller size).

- Later, we will use TCP segment while no segmentation in UDP.

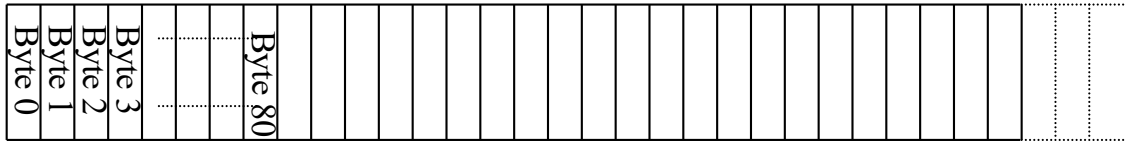
TCP "Stream of Bytes" Service

Host A

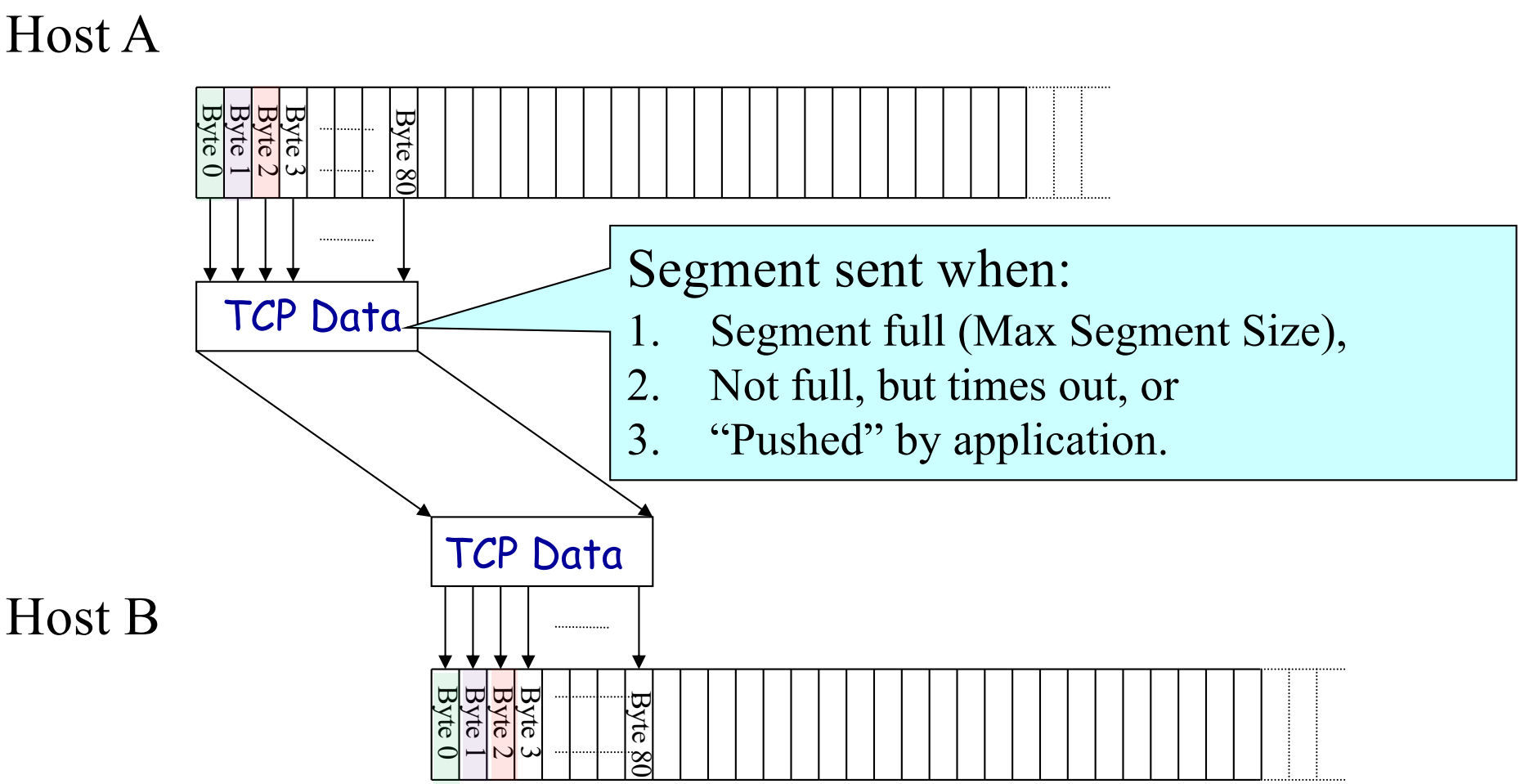


Although "byte stream" describes the service TCP offers to application processes, TCP does not, itself, transmit individual bytes over the Internet

Host B



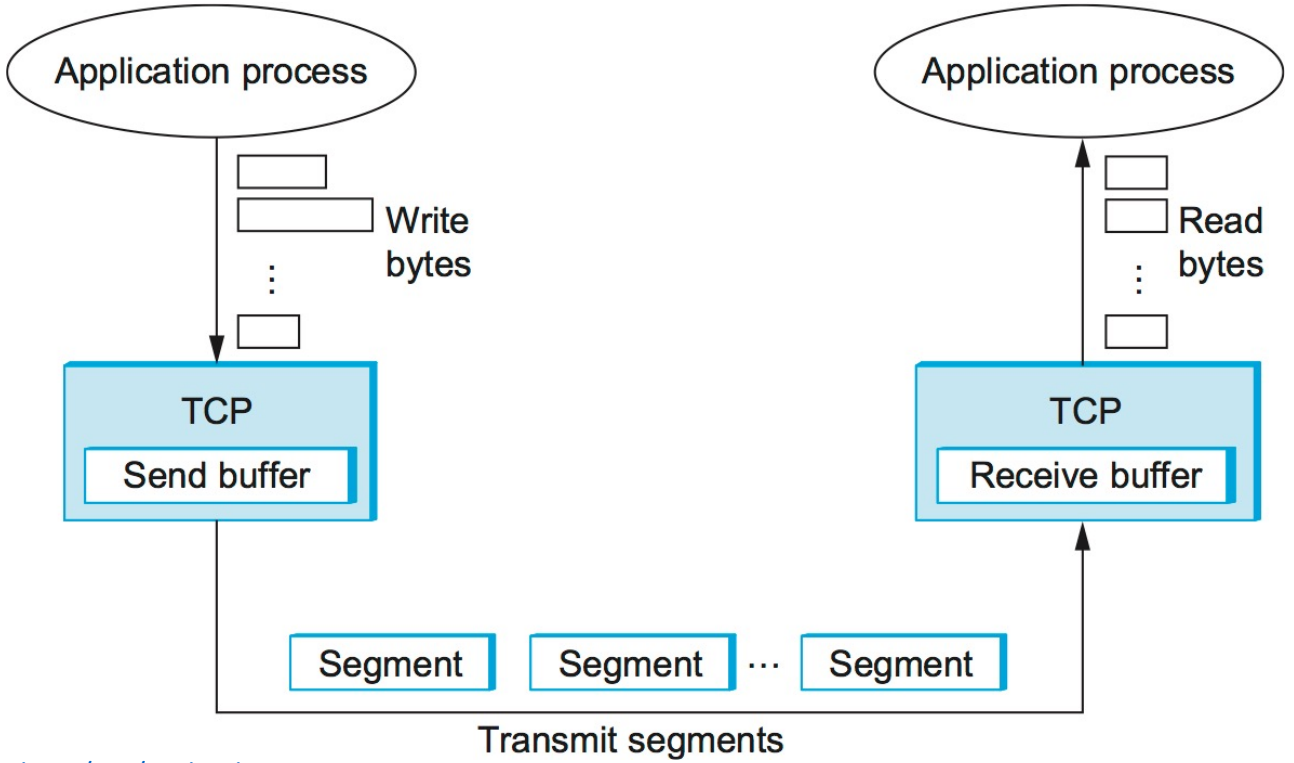
…Emulated Using TCP "Segments"



NOTE: Size of "TCP data" will be discussed later.

Segment transmission

TCP on the source host buffers enough bytes from the sending process to fill a reasonably sized packet and then sends this packet to its peer on the destination host. TCP on the destination host then empties the contents of the packet into a receive buffer, and the receiving process reads from this buffer at its leisure



https://book.systemsapproach.org/e2e/tcp.html

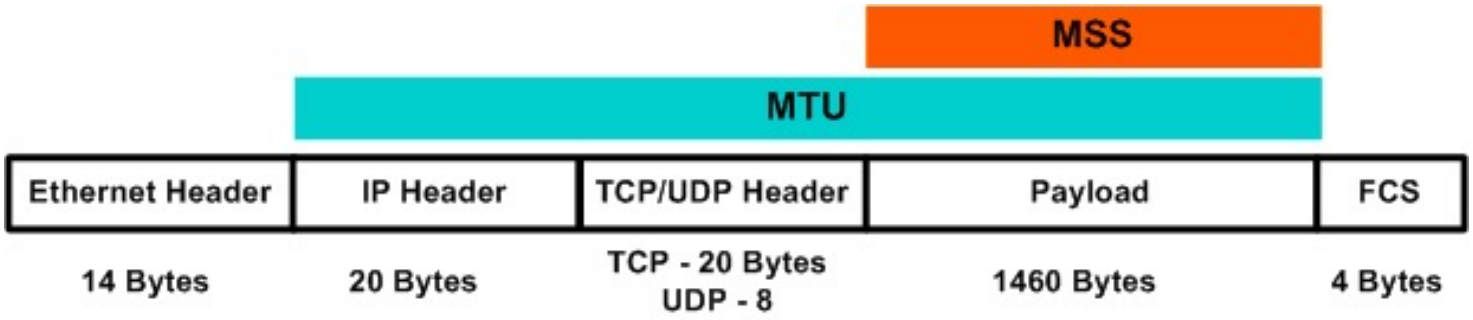
MTU and MSS

- Maximum Transmission Unit (MTU)
  - The MTU is how large the payload can be that is transferred over a link.
  - Standards can fix the size of an MTU or systems may decide MTU at connect time.
  - **For Example:** with Ethernet, the maximum frame size is 1518 bytes, 18 bytes of which are overhead (header and FCS), resulting in an MTU of 1500 bytes

7	Application Layer
6	Presentation Layer
5	Session Layer
4	Transport Layer
3	Network Layer
2	Datalink Layer
1	Physical Layer

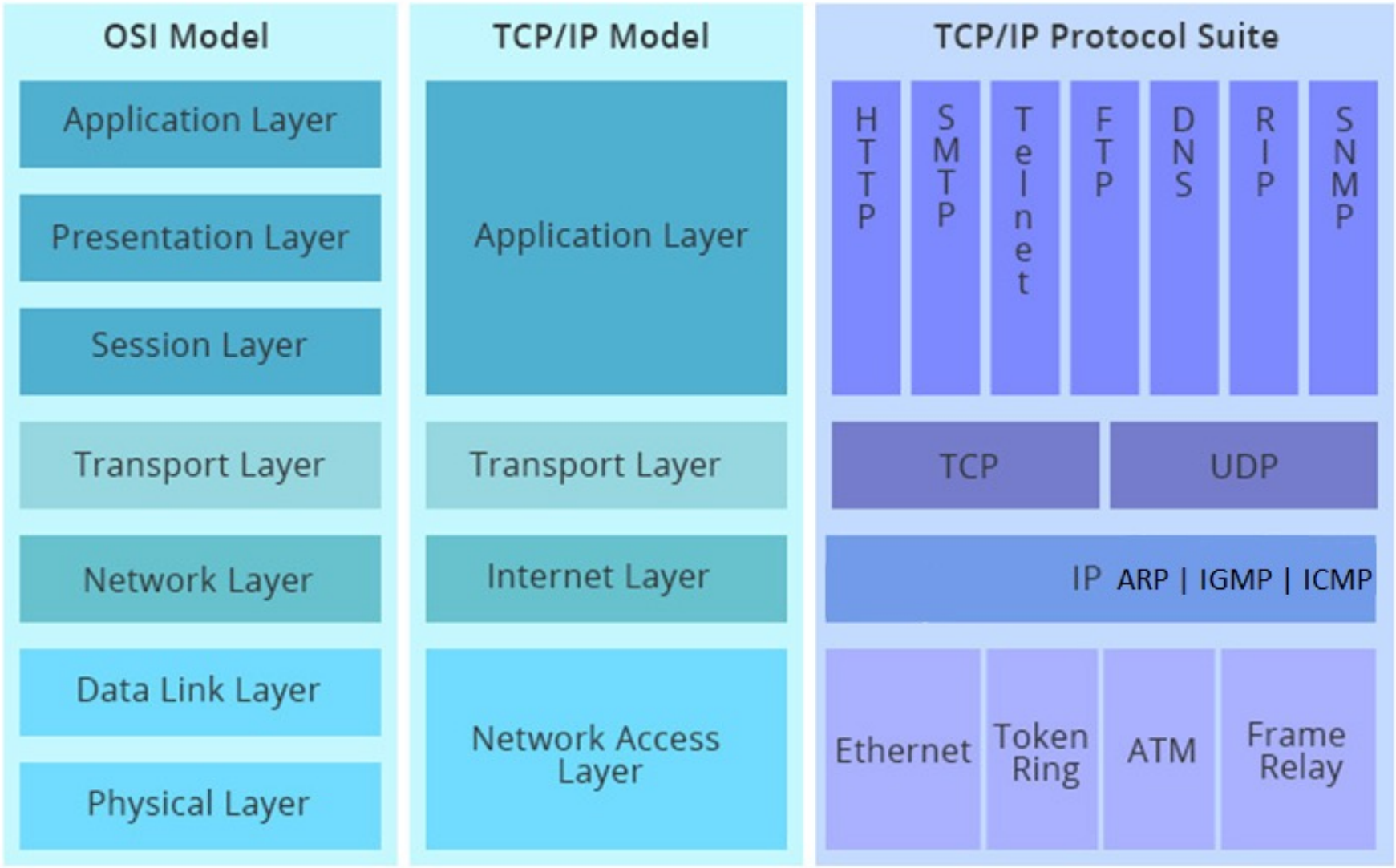
OSI Model

- Maximum Segment Size (MSS)
  - The MSS is used in TCP to signal how large payloads can be accepted when communicating.
  - TCP Maximum Segment Size is 536.
  - Where a host wishes to set the maximum segment size to a value other than the default, the maximum segment size is specified as a TCP option
  - The value cannot be changed after the connection is established



https://learningnetwork.cisco.com/thread/117909

TCP/IP Model vs OSI Model



https://community.fs.com/blog/tcpip-vs-osi-whats-the-difference-between-the-two-models.html

Segment vs. Datagram

- TCP *segment* is the protocol data unit which includes a TCP header and an application data piece (packet) which comes from the (upper) Application Layer.

- Transport layer data is generally named as *segment* and network layer data unit is named as *datagram*

- In case of UDP as transport layer protocol
  - We don't say UDP segment, instead, we say UDP *datagram* because we do not segment UDP data unit (segmentation is made in transport layer when we use TCP)

TCP vs UDP Header Format

TCP Segment Header Format

Bit #	0 – 7	8 – 15	16 – 23	24 – 31
0	Source Port		Destination Port	
32	Sequence Number			
64	Acknowledgment Number			
96	Data Offset	Res	Flags	Window Size
128	Header and Data Checksum		Urgent Pointer	
160...	Options			

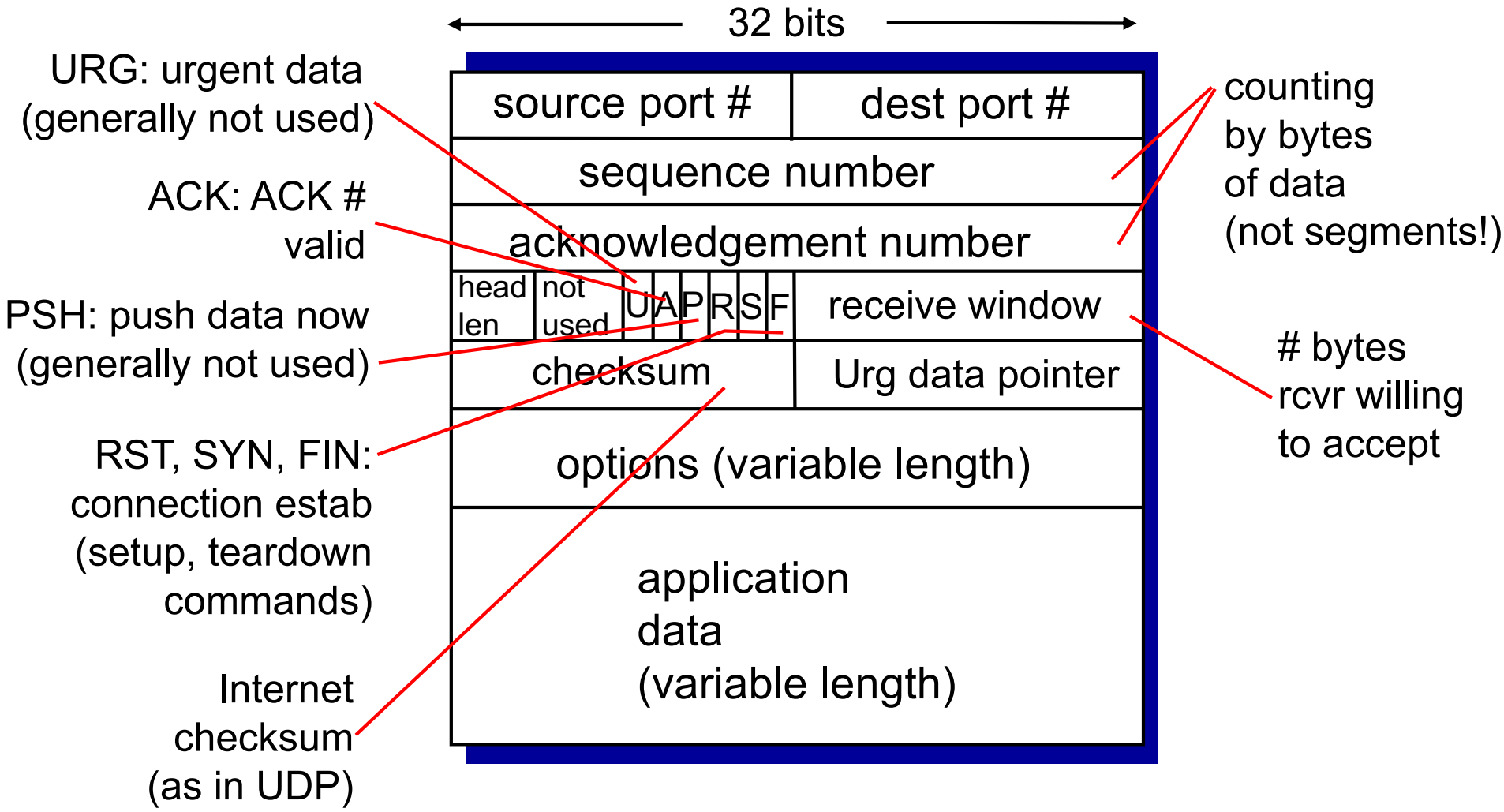
UDP Datagram Header Format

Bit #	0 – 7	8 – 15	16 – 23	24 – 31
0	Source Port		Destination Port	
32	Length		Header and Data Checksum	

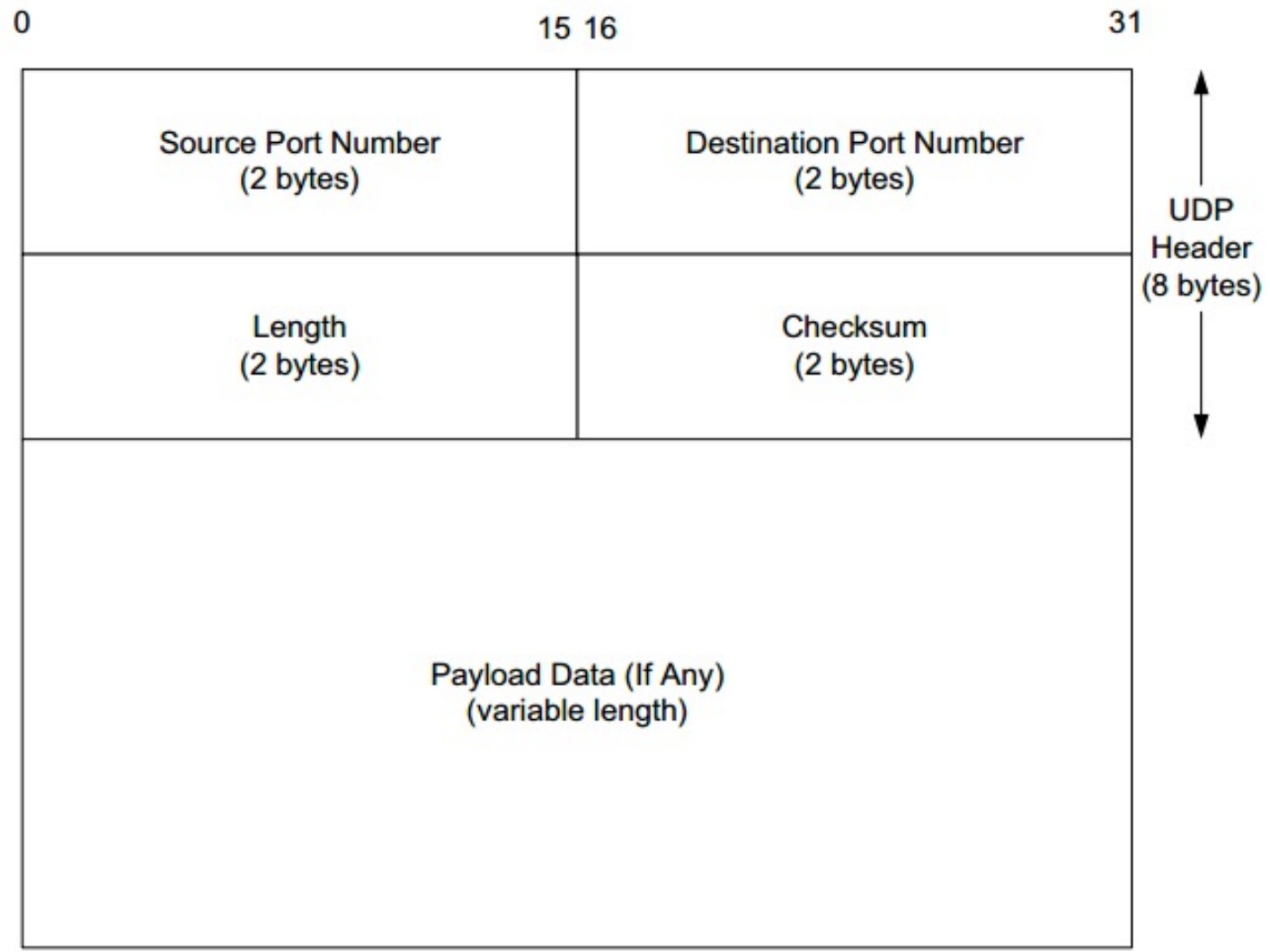
- Details will be discussed a little later

Source: https://skminhaj.wordpress.com/2016/02/15/tcp-segment-vs-udp-datagram-header-format/

TCP Segment Structure



UDP Datagram Structure



Source: https://notes.shichao.io/tcpv1/ch10/

Example Source code for TCP/IP client (1/2)

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include<unistd.h>


//Create a Socket for server communication
short SocketCreate(void){
	short hSocket;
	printf("Create the socket\n");
	hSocket = socket(AF_INET, SOCK_STREAM, 0);
	return hSocket;
}

//try to connect with server
int SocketConnect(int hSocket){
	int iRetval=-1;
	int ServerPort = 90190;
	struct sockaddr_in remote={0};

	remote.sin_addr.s_addr = inet_addr("127.0.0.1"); //Local Host
	remote.sin_family = AF_INET;
	remote.sin_port = htons(ServerPort);

	iRetval = connect(hSocket , (struct sockaddr *)&remote , sizeof(struct
sockaddr_in));

	return iRetval;
}
```

```
// Send the data to the server and set the timeout of 20 seconds
int SocketSend(int hSocket,char* Rqst,short lenRqst)  {

	int shortRetval = -1;
	struct timeval tv;
	tv.tv_sec = 20;  /* 20 Secs Timeout */
	tv.tv_usec = 0;

	if(setsockopt(hSocket, SOL_SOCKET, SO_SNDTIMEO, (char *)&tv,sizeof(tv)) < 0){
	  printf("Time Out\n");
	  return -1;}
	shortRetval = send(hSocket , Rqst , lenRqst , 0);

	return shortRetval;
}

//receive the data from the server
int SocketReceive(int hSocket,char* Rsp,short RvcSize)  {

int shortRetval = -1;
struct timeval tv;
tv.tv_sec = 20;  /* 20 Secs Timeout */
tv.tv_usec = 0;

if(setsockopt(hSocket, SOL_SOCKET, SO_RCVTIMEO, (char *)&tv,sizeof(tv)) < 0)  {
  printf("Time Out\n");
  return -1;
}
shortRetval = recv(hSocket, Rsp , RvcSize , 0);

printf("Response %s\n",Rsp);

return shortRetval;
}
```

Example Source code for TCP/IP client (2/2)

```
//main driver program
int main(int argc , char *argv[])  {
int hSocket, read_size;
struct sockaddr_in server;
char SendToServer[100] = {0};
char server_reply[200] = {0};

//Create socket
hSocket = SocketCreate();
if(hSocket == -1)  {
  printf("Could not create socket\n");
  return 1;
}

printf("Socket is created\n");

//Connect to remote server
if (SocketConnect(hSocket) < 0)
{
perror("connect failed.\n");
return 1;
}

printf("Sucessfully connected with server\n");

printf("Enter the Message: ");
gets(SendToServer);
```

```
//Send data to the server
SocketSend(hSocket , SendToServer , strlen(SendToServer));


   //Received the data from the server
read_size = SocketReceive(hSocket , server_reply , 200);
printf("Server Response : %s\n\n",server_reply);


close(hSocket);
shutdown(hSocket,0);
shutdown(hSocket,1);
shutdown(hSocket,2);
   return 0;
}
```

Example Source code for TCP/IP server (1/2)

```
#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include<unistd.h>


short SocketCreate(void){
short hSocket;
printf("Create the socket\n");

hSocket = socket(AF_INET, SOCK_STREAM, 0);
return hSocket;
}


int BindCreatedSocket(int hSocket){
int iRetval=-1;
int ClientPort = 90190;
struct sockaddr_in  remote={0};


remote.sin_family = AF_INET; /* Internet address family */
remote.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
remote.sin_port = htons(ClientPort); /* Local port */
iRetval = bind(hSocket,(struct sockaddr *)&remote,sizeof(remote));

      return iRetval;
}
```

```
int main(int argc , char *argv[]){
int socket_desc , sock , clientLen , read_size;
struct sockaddr_in server , client;
char client_message[200]={0};
char message[100] = {0};
const char *pMessage = "hello aticleworld.com";

//Create socket
socket_desc = SocketCreate();
if (socket_desc == -1){
printf("Could not create socket");
return 1;
}
printf("Socket created\n");

//Bind
if( BindCreatedSocket(socket_desc) < 0){
//print the error message
perror("bind failed.");
return 1;
}
printf("bind done\n");

//Listen
listen(socket_desc , 3);
```

Example Source code for TCP/IP server (2/2)

```
//Accept and incoming connection

while(1){
printf("Waiting for incoming connections...\n");
clientLen = sizeof(struct sockaddr_in);

//accept connection from an incoming client
sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t*)&clientLen);
if (sock < 0){
perror("accept failed");
return 1;
}
printf("Connection accepted\n");


memset(client_message, '\0', sizeof client_message);
memset(message, '\0', sizeof message);
//Receive a reply from the client
if( recv(sock , client_message , 200 , 0) < 0){
printf("recv failed");
break;
}
printf("Client reply : %s\n",client_message);


if(strcmp(pMessage,client_message)==0){
strcpy(message,"Hi there !");
}
Else{
  strcpy(message,"Invalid Message !");
}
```

```
// Send some data
if( send(sock , message , strlen(message) , 0) < 0)
{
printf("Send failed");
return 1;
}

close(sock);
sleep(1);
}
   return 0;
}
```

UDP Server – Code Example

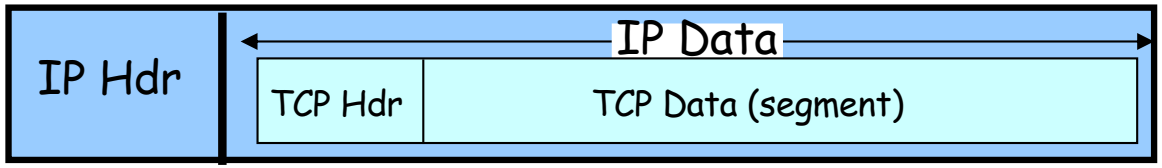
```c
// Server side implementation of UDP client-server model
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT     8080
#define MAXLINE 1024

// Driver code
int main() {
    int sockfd;
    char buffer[MAXLINE];
    char *hello = "Hello from server";
    struct sockaddr_in servaddr, cliaddr;

    // Creating socket file descriptor
    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&servaddr, 0, sizeof(servaddr));
    memset(&cliaddr, 0, sizeof(cliaddr));

    // Filling server information
    servaddr.sin_family    = AF_INET; // IPv4
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(PORT);
```

```c
    // Bind the socket with the server address
    if ( bind(sockfd, (const struct sockaddr *)&servaddr,
            sizeof(servaddr)) < 0 )
    {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    int len, n;
    n = recvfrom(sockfd, (char *)buffer, MAXLINE,
                MSG_WAITALL, ( struct sockaddr *) &cliaddr,
                &len);
    buffer[n] = '\0';
    printf("Client : %s\n", buffer);
    sendto(sockfd, (const char *)hello, strlen(hello),
        MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
            len);
    printf("Hello message sent.\n");

    return 0;
}
```

UDP Client – Code Example

```c
// Client side implementation of UDP client-server model
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT     8080
#define MAXLINE 1024

// Driver code
int main() {
    int sockfd;
    char buffer[MAXLINE];
    char *hello = "Hello from client";
    struct sockaddr_in     servaddr;

    // Creating socket file descriptor
    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&servaddr, 0, sizeof(servaddr));

    // Filling server information
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = INADDR_ANY;

    int n, len;

    sendto(sockfd, (const char *)hello, strlen(hello),
        MSG_CONFIRM, (const struct sockaddr *) &servaddr,
            sizeof(servaddr));
    printf("Hello message sent.\n");

    n = recvfrom(sockfd, (char *)buffer, MAXLINE,
                MSG_WAITALL, (struct sockaddr *) &servaddr,
                &len);
    buffer[n] = '\0';
    printf("Server : %s\n", buffer);

    close(sockfd);
    return 0;
}
```

TCP Segment



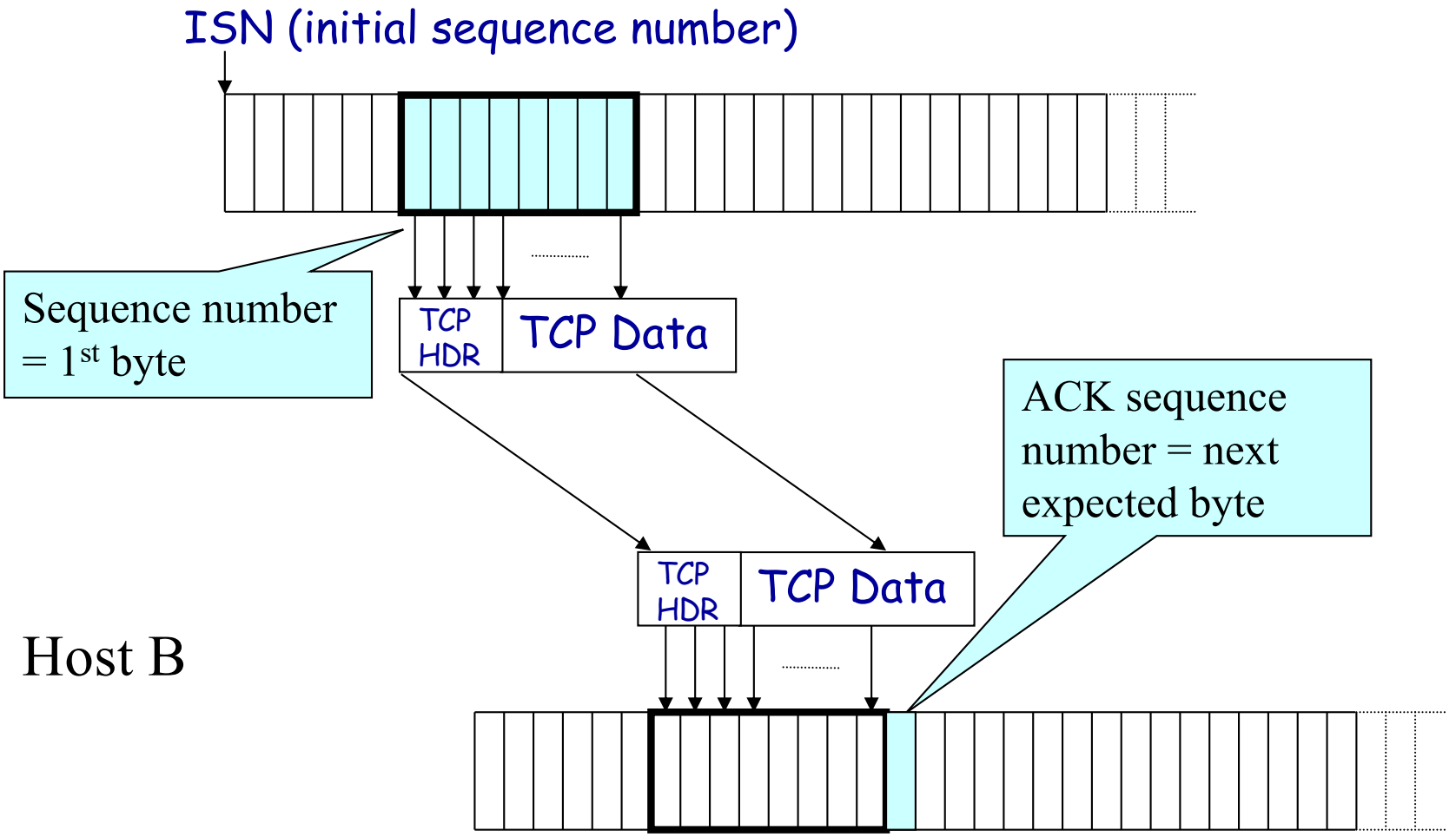
- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes on an Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header is typically 20 bytes long
- TCP segment
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream
- TCP header
  - TCP Hdr offset is 0 that is the same as in data structure.

Initial Sequence Number (ISN)

- Sequence number for the very first byte
  - E.g., Why not a de facto ISN of 0?
- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - … and there is a chance an old packet is still in flight
  - … and might be associated with the new connection
- So, TCP requires changing the ISN over time
  - Set from a 32-bit clock that ticks every 4 microseconds
  - … which only wraps around once every 4.55 hours!
- But, this means the hosts need to exchange ISNs

Sequence Numbers

Host A



Host B

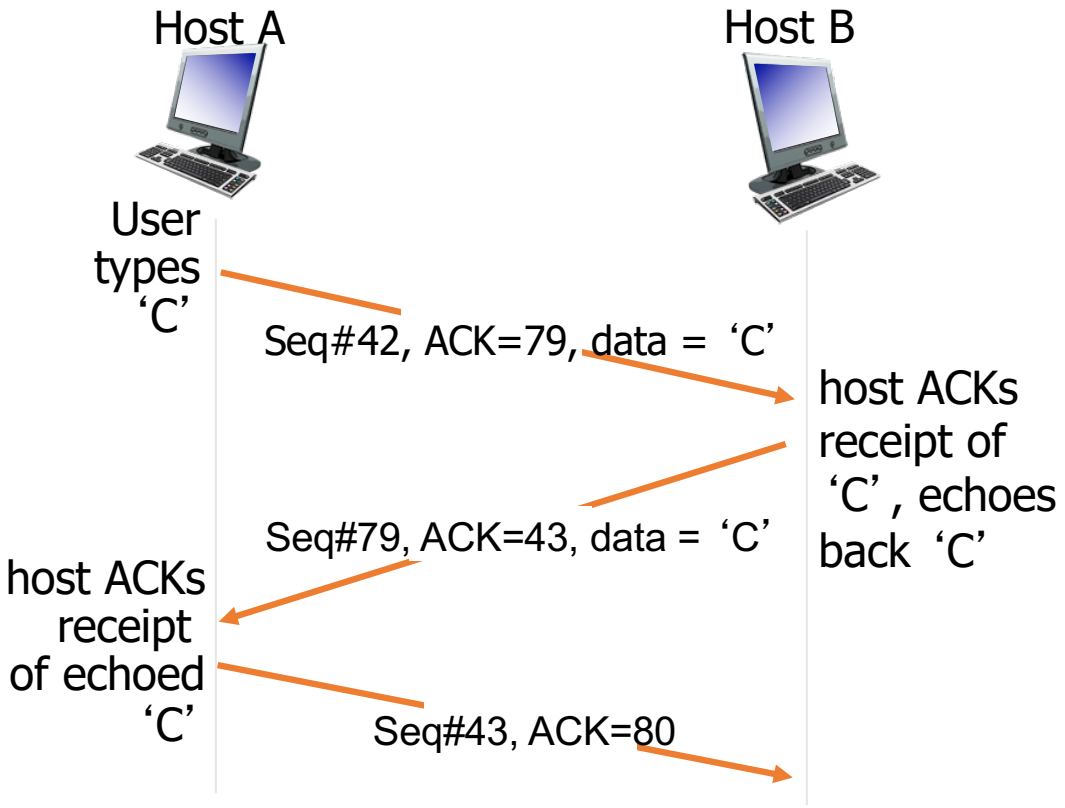
TCP Sequence numbers, ACKs

- **Example:**
  - Host A sends a character to Host B, which echoes it back to Host A.
  - Starting Sequence# for client and server are 42 and 79

- **Seq. #'s:**
  - byte stream "number" of first byte in segment's data

- **ACKs:**
  - Seq# of next byte expected from other side
  - cumulative ACK

- **Q:** how receiver handles out-of-order segments?
- **A:** TCP spec doesn't say, - up to implementor



Simple Telnet Scenario

More details: https://www.net.t-labs.tu-berlin.de/teaching/computer_networking/03.05.htm

TCP round trip time, timeout

- Round Trip Time (RTT) is the length of time it takes for a signal to be sent plus the length of time it takes for an acknowledgement of that signal to be received.
  - This time delay includes the propagation times for the paths between the two communication endpoints.

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

Q: how to estimate RTT?

- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current **`SampleRTT`**

NOTE: Do not mess with TTL

Topic of the lecture

- Connection-oriented transport: TCP
  - Connection management
  - Segment structure
  - Reliable data transfer
  - Flow control
- Principles of congestion control
- TCP congestion control

TCP Reliable Data Transfer

- TCP creates rdt service on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer

- Retransmissions triggered by:
  - timeout events
  - duplicate acks

- Let's initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

TCP Timeout & Retransmission

- TCP provides a reliable transport layer.
- One of the ways it provides reliability is for each end to acknowledge the data it receives from the other end.
- But data segments and acknowledgments can get lost.
- TCP handles this by setting a timeout when it sends data, and if the data isn't acknowledged when the timeout expires, it retransmits the data.

TCP Sender Events

Data received from app:

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: **`TimeOutInterval`**

Timeout:

- retransmit segment that caused timeout
- restart timer

ACK received

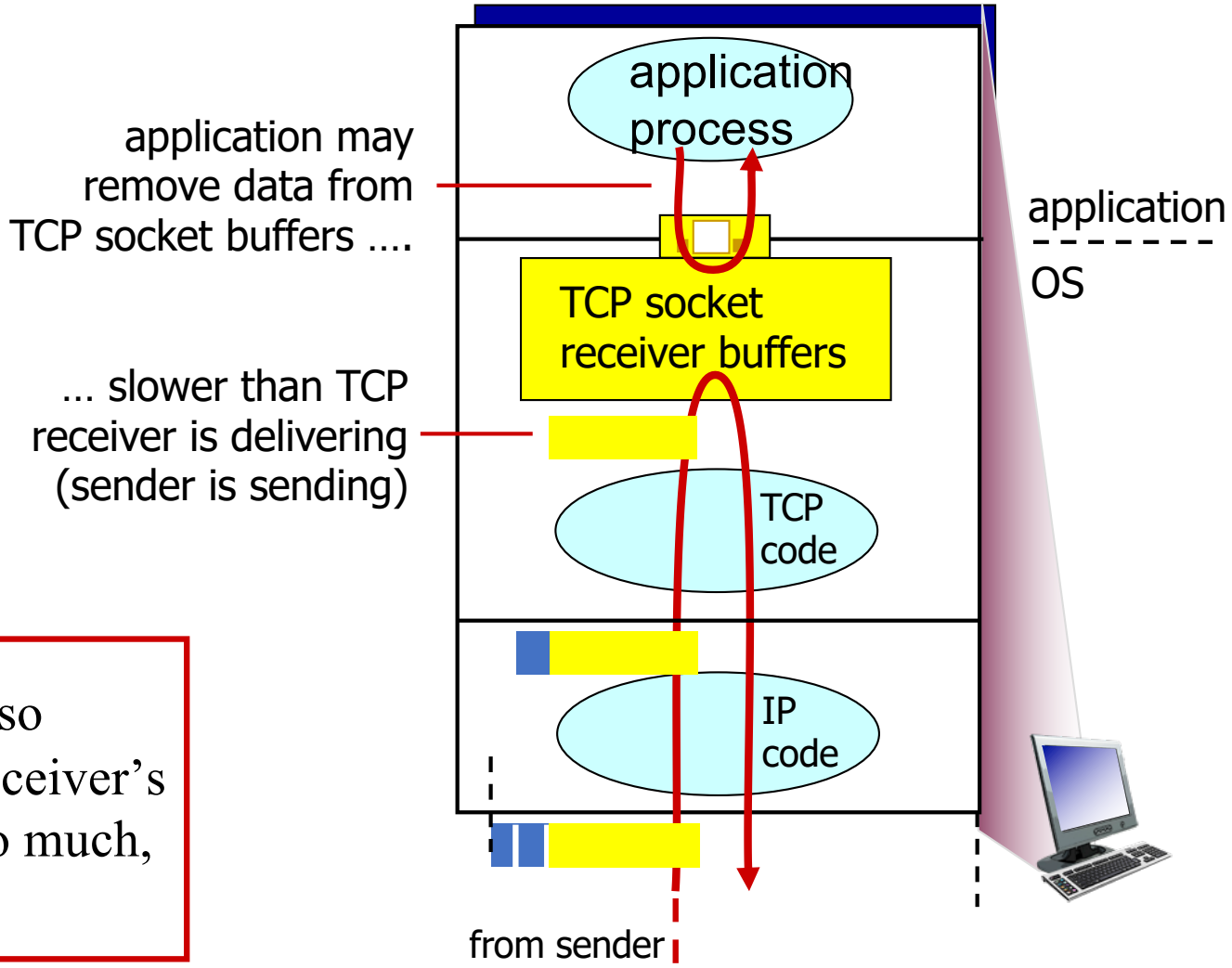
- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

NOTE: Terminate the previous connection (if any)

Topic of the lecture

- Connection-oriented transport: TCP
  - Connection management
  - Segment structure
  - Reliable data transfer
  - Flow control
- Principles of congestion control
- TCP congestion control

TCP Flow Control

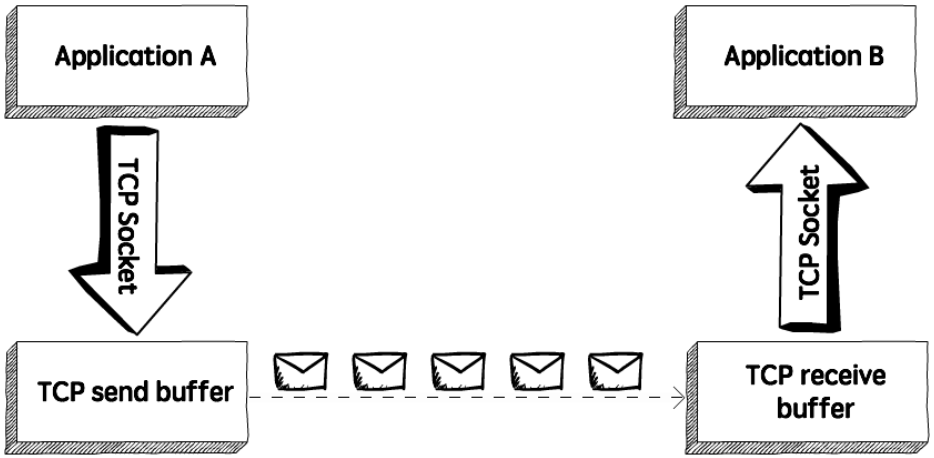


Receiver Protocol Stack

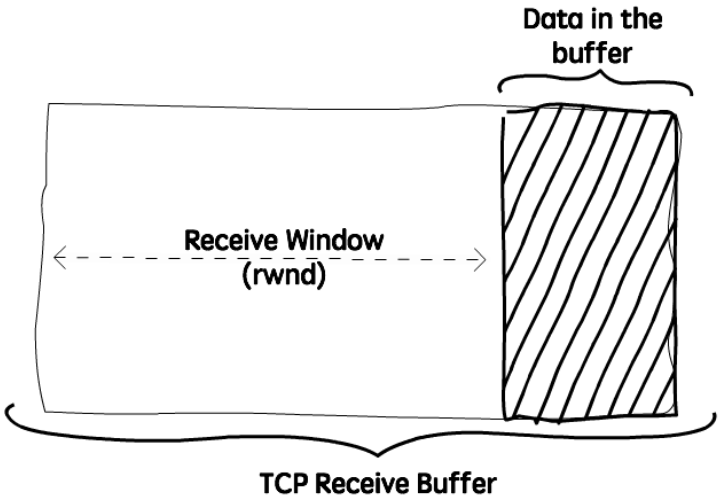
Flow control

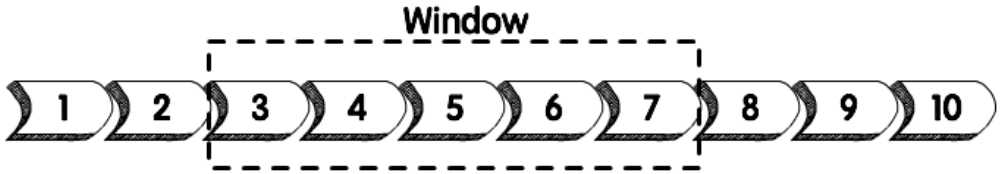
receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

TCP Flow Control



- Lot of ways to achieve the high performance
  - Simplest method: Buffer
  - Multiple connection for single file
  - TCP/IP tailoring

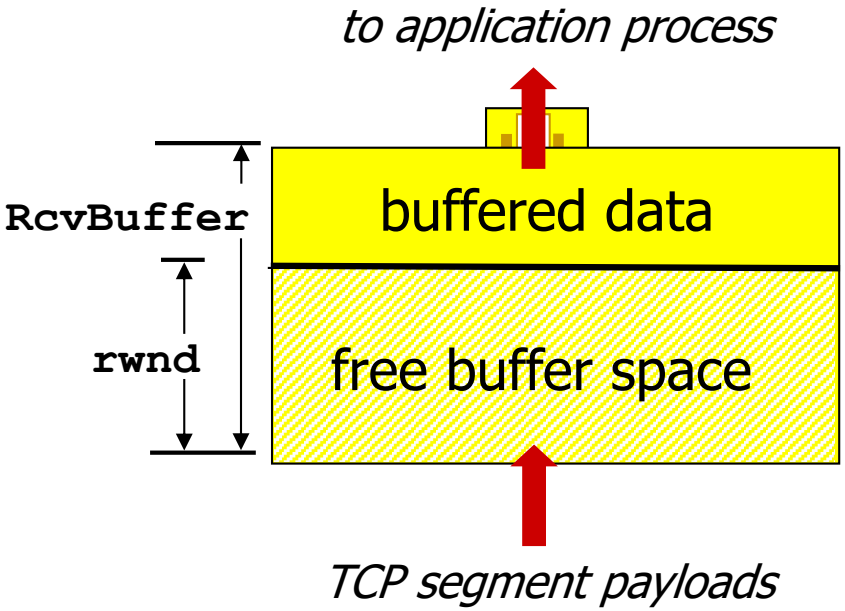




Source: https://www.brianstorti.com/tcp-flow-control/

TCP Flow Control

- Receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- Sender limits amount of unacked ("in-flight") data to receiver's **`rwnd`** value
- Guarantees receive buffer will not overflow



Receiver-side buffering

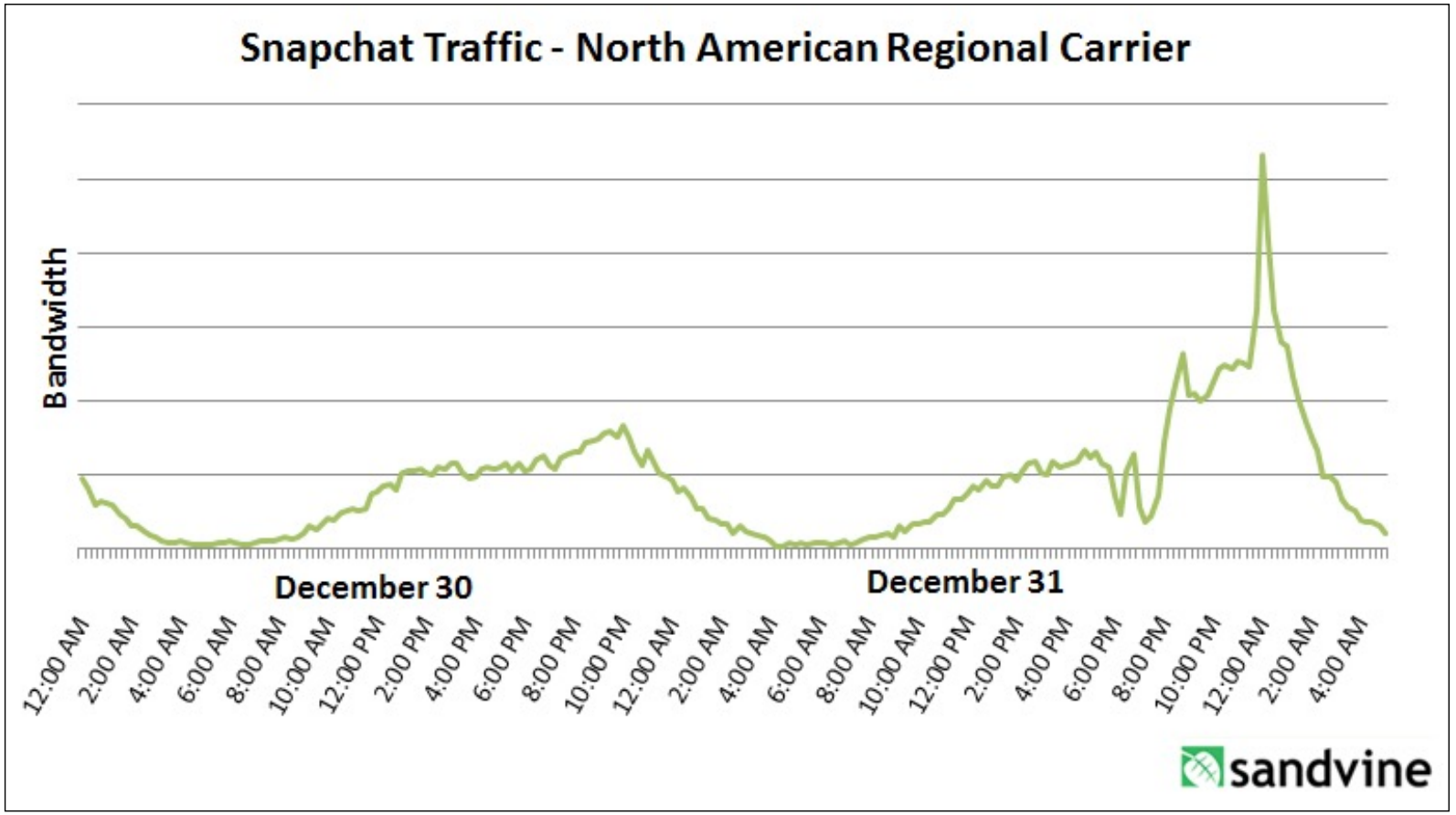
Topic of the lecture

- Connection-oriented transport: TCP
  - Connection management
  - Segment structure
  - Reliable data transfer
  - Flow control
- Principles of congestion control
- TCP congestion control

What is Congestion?

- Load on the network is higher than capacity
  - Capacity is not uniform across networks
    - Modem vs. Cellular vs. Cable vs. Fiber Optics
  - There are multiple flows competing for bandwidth
    - Residential cable modem vs. corporate datacenter
  - Load is not uniform over time
    - 10pm, Sunday night = Bittorrent Game of Thrones
  - Typically load of network fluctuates
    - During the Christmas holiday

What is Congestion?



Why is Congestion Bad?

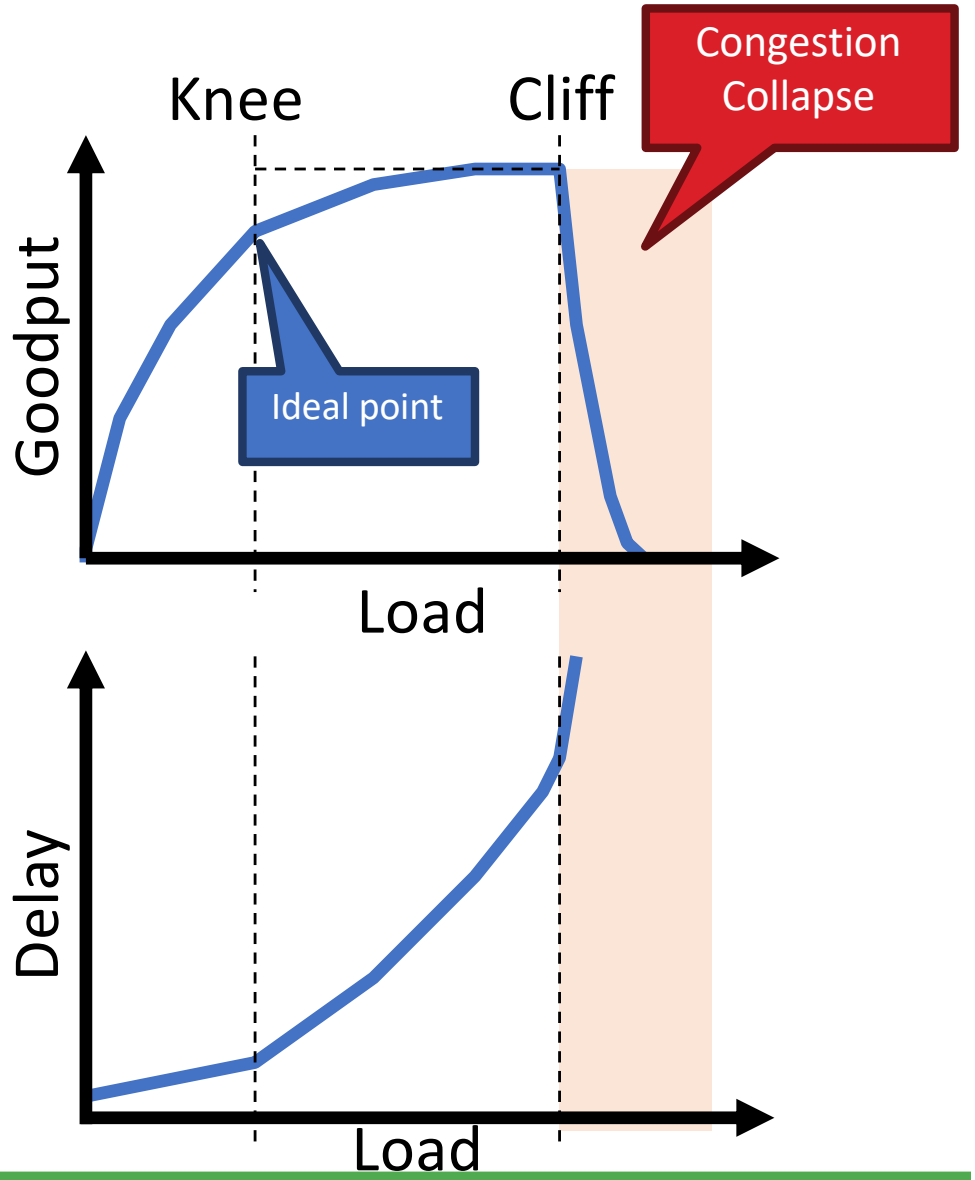
- Some of the hardware can be physically damaged
- Results in packet loss
  - Routers have finite buffers
  - Internet traffic is self similar, no buffer can prevent all drops
  - When routers get overloaded, packets will be dropped
- Practical consequences
  - Router queues build up, delay increases
  - Wasted bandwidth from retransmissions
  - Low network “goodput”

What is a Congestion Window?

- TCP uses a congestion window in the sender side to do congestion avoidance.
- The congestion window indicates the maximum amount of data that can be sent out on a connection without being acknowledged.
- TCP detects congestion when it fails to receive an acknowledgement for a packet within the estimated timeout.

The Danger of Increasing Load

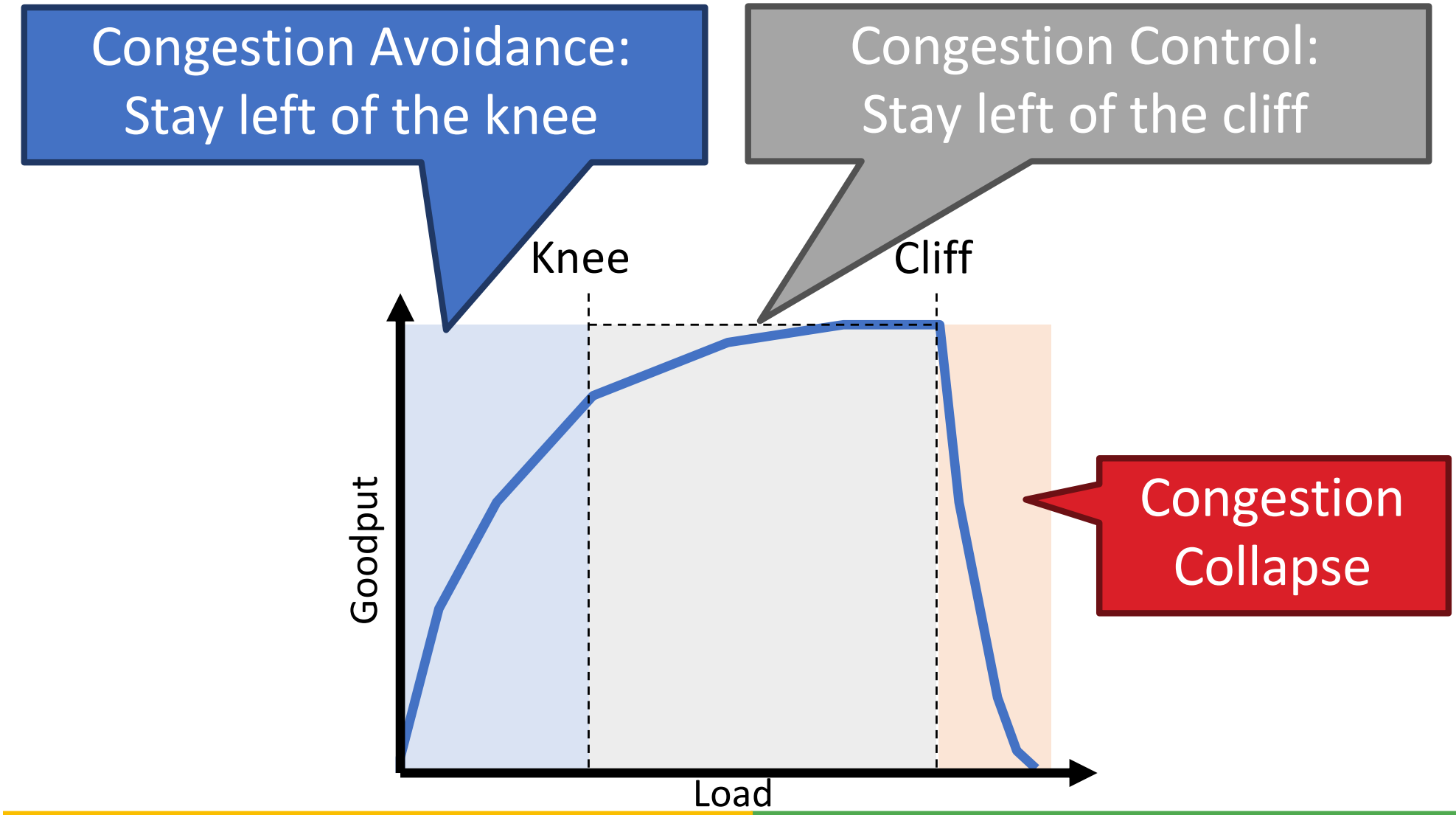
- Knee – point after which
  - Throughput increases very slow
  - Delay increases fast
- In an M/M/1 queue
  - Delay = 1/(1 – utilization)
- Cliff – point after which
  - Throughput → 0
  - Delay → ∞



Congestion Control

- When a new connection is established with a host on another network, the congestion window is initialized to one segment
  - The segment size announced by the other end, or the default, typically 536 or 512).
- Each time an ACK is received, the congestion window is increased by one segment.
- The sender can transmit up to the minimum of the congestion window and the advertised window.
- The congestion window is flow control imposed by the sender, while the advertised window is flow control imposed by the receiver.

Cong. Control vs. Cong. Avoidance



Advertised Window, Revisited

- Does TCP's advertised window solve congestion?

  NO

- The advertised window only protects the receiver

- A sufficiently fast receiver can max the window
  - What if the network is slower than the receiver?
  - What if there are other concurrent flows?

- Key points
  - Window size determines send rate
  - Window must be adjusted to prevent congestion collapse

Goals of Congestion Control

1. Adjusting to the bottleneck bandwidth
2. Adjusting to variations in bandwidth
3. Sharing bandwidth between flows
4. Maximizing throughput

General Approaches

- Do nothing, send packets indiscriminately
  - Many packets will drop, totally unpredictable performance
  - May lead to congestion collapse
- Reservations
  - Pre-arrange bandwidth allocations for flows
  - Requires negotiation before sending packets
  - Must be supported by the network
- Dynamic adjustment
  - Use probes to estimate level of congestion
  - Speed up when congestion is low
  - Slow down when congestion increases
  - Messy dynamics, requires distributed coordination

Congestion Collapse

- Congestive collapse (or congestion collapse) is the condition in which congestion prevents or limits useful communication.
- Congestion collapse generally occurs at choke points in the network, where incoming traffic exceeds outgoing bandwidth.
- Connection points between a local area network and a wide area network are common choke points.
- When a network is in this condition, it settles into a stable state where traffic demand is high but little useful throughput is available, packet delay and loss occur and quality of service is extremely poor.
- **Solution**
  - Nagle's algorithm used for improving the efficiency of TCP/IP networks by reducing the number of packets that need to be sent over the network.
- NOTE: Nagle's algorithm is one of the possible solutions. Many other also exists…

Topic of the lecture

- Connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
- Principles of congestion control
- TCP congestion control

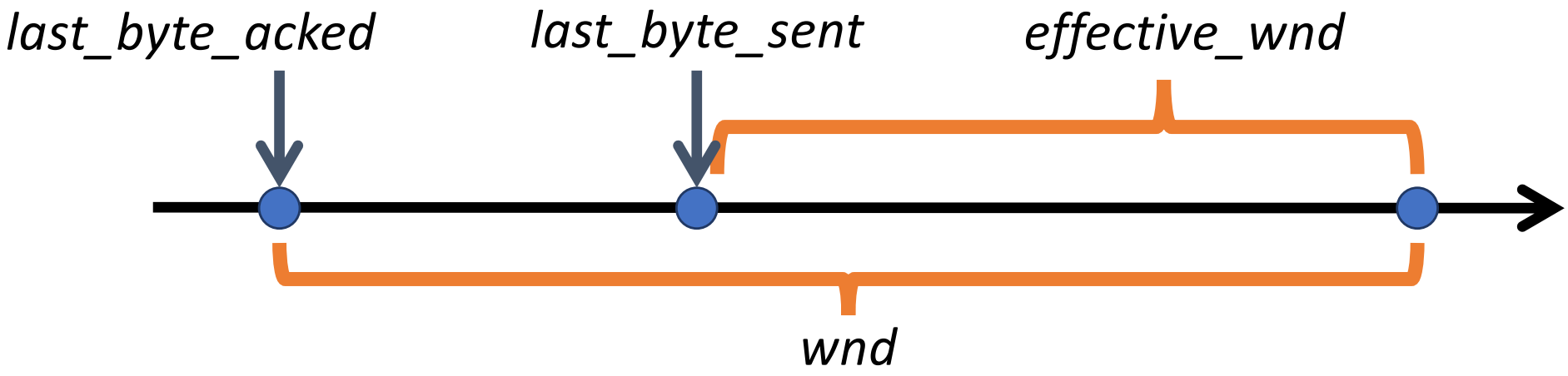
TCP Congestion Control

- Each TCP connection has a window
  - Controls the number of unACKed packets
- Sending rate is ~ window/RTT
- Idea: vary the window size to control the send rate
- Introduce a congestion window at the sender
  - Congestion control is sender-side problem

Congestion Window (*cwnd*)

- Limits how much data is in transit
- Denominated in bytes

1. *wnd* = min(*cwnd*, *adv_wnd*);
2. *effective_wnd* = *wnd* –
   (*last_byte_sent* – *last_byte_acked*);



Two Basic Components

1. Detect congestion
   - Packet dropping is most reliably signal
     - Delay-based methods are hard and risky
   - How do you detect packet drops? ACKs
     - Timeout after not receiving an ACK
     - Several duplicate ACKs in a row (ignore for now)

2. Rate adjustment algorithm
   - Modify *cwnd*
   - Probe for bandwidth
   - Responding to congestion

INNOPOLIS UNIVERSITY

Rate Adjustment

- Recall: TCP is ACK clocked
  - Congestion = delay = long wait between ACKs
  - No congestion = low delay = ACKs arrive quickly

- Basic algorithm
  - Upon receipt of ACK: increase cwnd
    - Data was delivered, perhaps we can send faster
    - *cwnd* growth is proportional to RTT
  - On loss: decrease cwnd
    - Data is being lost, there must be congestion

Summary

- Connection-oriented transport: TCP
  - Connection management
  - Segment structure
  - Reliable data transfer
  - Flow control
- Principles of congestion control
- TCP congestion control