Communication Technologies for Critical Systems
Networking Concepts Review

Mestrado em Engenharia de Sistemas Computacionais Críticos

Network

- Set of computers sharing resources located on or provided by network nodes enabled by communicating between them.
- Communications use digital interconnections to communicate with each other.
- These interconnections are made up of telecommunication network technologies, based on physically wired, optical, and wireless radio-frequency methods that may be arranged in a variety of network topologies.

Types of Networks

- Local Area Networks (LAN)
  - LANs connect groups of computers and low-voltage devices together across short distances (within a building or between a group of two or three buildings in close proximity to each other). Enterprises typically manage and maintain LANs.
  - Using routers, LANs can connect to wide area networks (WANs, explained below) to rapidly and safely transfer data.
- Metropolitan Area Networks (MAN)
  - Larger than LANs but smaller than WANs – incorporate elements from both types of networks.
  - MANs span an entire geographic area (typically a town or city, but sometimes a campus). Ownership and maintenance is handled by either a single person or company (a local council, a large company, etc.).

Types of Networks

- Wide Area Networks (WAN)
  - Connects computers together across longer physical distances. The Internet is the most basic example of a WAN. It is typically owned and maintained by multiple administrators or the public.
- Personal Area Network (PAN)
  - made up of a wireless modem, a computer or two, phones, printers, tablets, etc., and revolves around one person in one building. These types of networks are typically found in small offices or residences, and are managed by one person or organization from a single device.

Types of Networks

- Body Area Networks (BAN)
  - is the interconnection of multiple computing devices worn on, affixed to or implanted in a person's body. Typically includes a smartphone that serves as a mobile data hub, acquiring user data and transmitting it to a remote database or other system
- Storage-Area Network (SAN)
  - Dedicated high-speed network that connects shared pools of storage devices to several servers, these types of networks don’t rely on a LAN or WAN. Instead, they move storage resources away from the network and place them into their own high-performance network. SANs can be accessed in the same fashion as a drive attached to a server.

Types of Networks

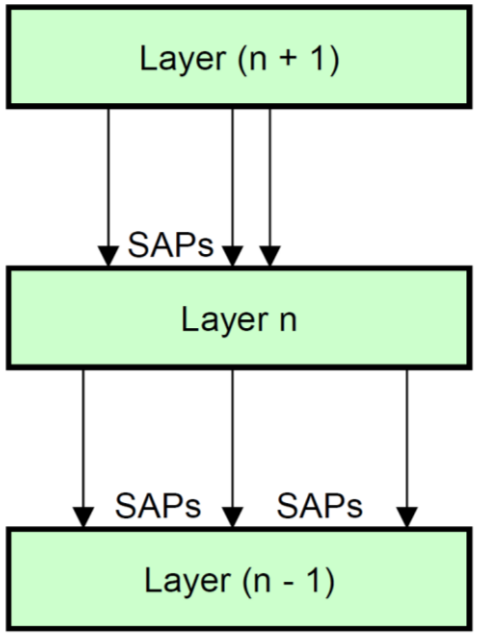
- Virtual Private Network (VPN)
  - By extending a private network across the Internet, a VPN lets its users send and receive data as if their devices were connected to the private network – even if they're not. Through a virtual point-to-point connection, users can access a private network remotely.
  - It protects communications by encryption and authentication methods

The OSI Model

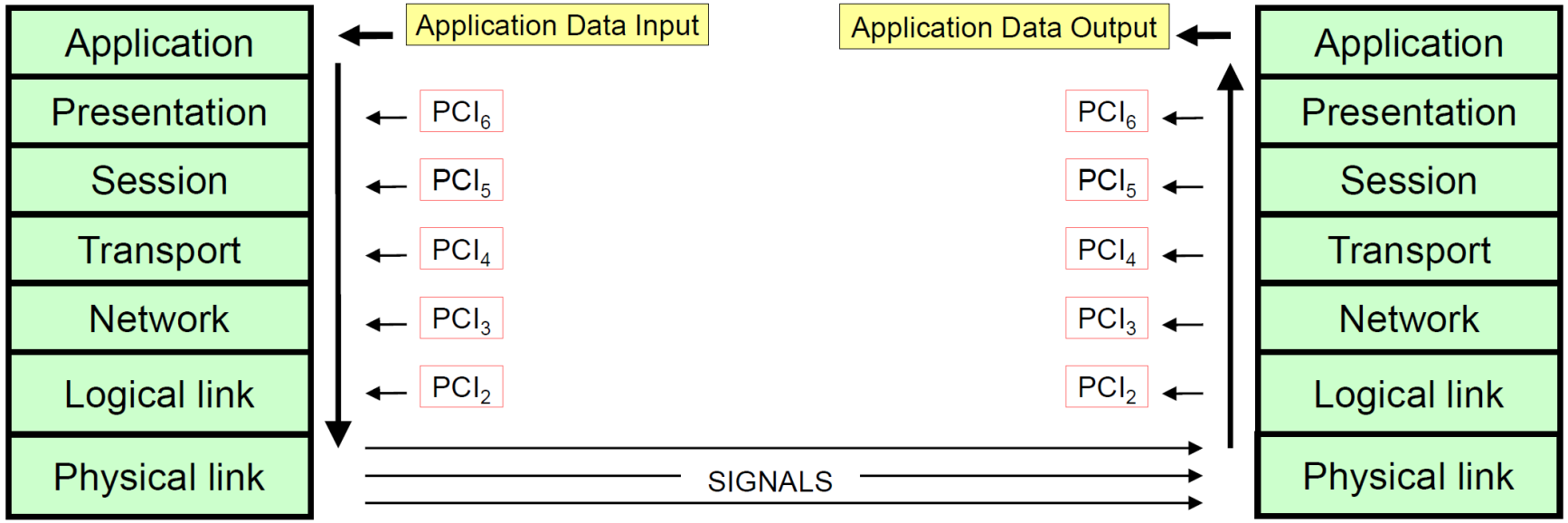
7	Application Layer	Human-computer interaction layer, where applications can access the network services
6	Presentation Layer	Ensures that data is in a usable format and is where data encryption occurs
5	Session Layer	Maintains connections and is responsible for controlling ports and sessions
4	Transport Layer	Transmits data using transmission protocols including TCP and UDP
3	Network Layer	Decides which physical path the data will take
2	Data Link Layer	Defines the format of data on the network
1	Physical Layer	Transmits raw bit stream over the physical medium

The OSI Model: Layers

- Successive layers of the stack interact with each other according to a downward service call model through **service access points** (SAP).
- Each layer only uses the services provided by layer below and adds them new features.
  - The new features implemented require the addition of control information (PCI **Protocol Control Information**).
  - The PCI is added to the data (SDU-**Service Data Unit**) coming from the upper layer, i.e., the payload.
  - The PCI and the SDU together make the PDU (**Protocol Data Unit**), i.e., the packet.



The OSI model: protocols



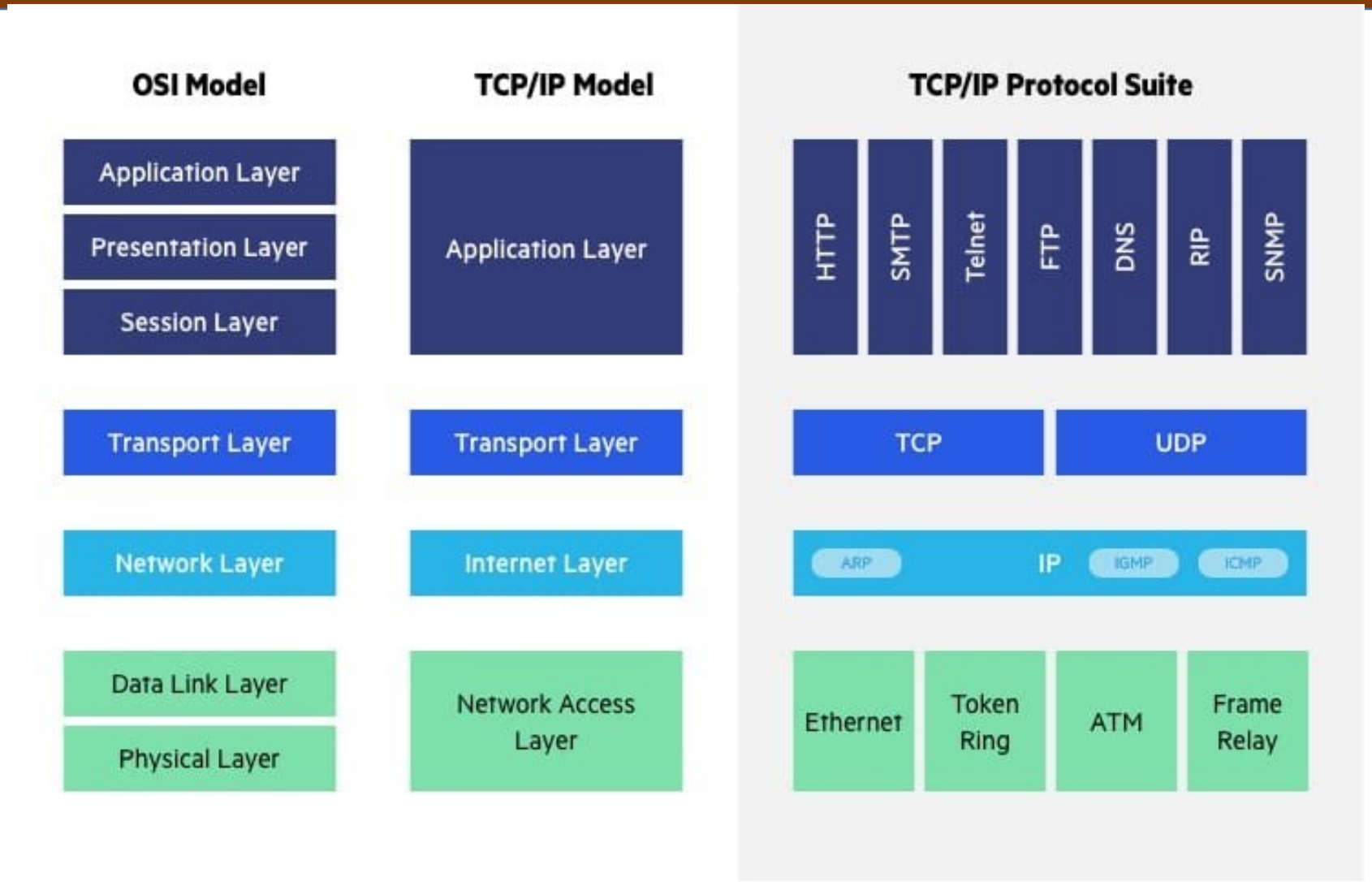
The role of the OSI reference architecture

- Beyond the architectural description:
  - OSI also tried to force some concrete protocol implementations for some layers.
  - However, most of such concrete implementations proposed in OSI were never widely adopted in real computer networks.
    - One reason for that is manufactures where not willing to give away their proprietary architectures and restart implementations from scratch
    - While under the point of view of concrete implementations it has not been a success,
  - The OSI model was a very important step because it includes a set of concepts, standards, nomenclature, techniques and ideas that have become a reference point for any discussion in the computer networks area.
  - Thus, it's often referred to as the **OSI reference model**

The role of the OSI reference architecture

- One OSI main motivation - to unite all network architectures to allow all nodes to talk to each other, was ultimately achieved by another architecture TCP/IP
- TCP/IP
  - Achieved it by becoming global through the internet, by doing so, once users everywhere start demanding an internet connection, manufacturers were compelled to adopt it

TCP/IP

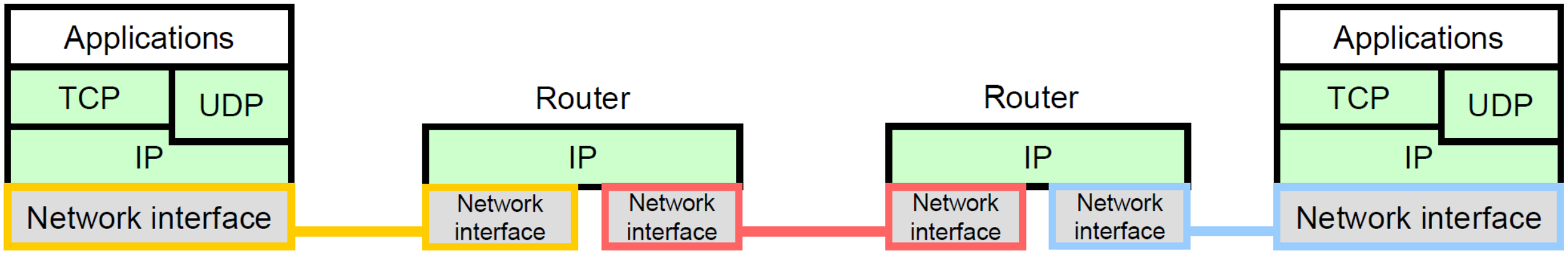

OSI Model
TCP/IP Model
TCP/IP Protocol Suite
Application Layer
Presentation Layer
Session Layer
Application Layer
HTTP
SMTP
Telnet
FTP
DNS
RIP
SNMP
Transport Layer
Transport Layer
TCP
UDP
Network Layer
Internet Layer
ARP
IP
IGMP
ICMP
Data Link Layer
Physical Layer
Network Access Layer
Ethernet
Token Ring
ATM
Frame Relay

Routing

- One advantage of TCP/IP is it doesn't define layer one neither layer two.
- The IP protocol is expected to use any existing layer two packet transmission technology.
- IP defines a universal packet format and universal node addressing, thus allowing the interconnection of different network types. This allows the building of a global network such as the internet.
- However, when an IP node uses some specific layer two technology it will be able to deliver only to other IP nodes directly connected to that same technology (To overcome this, **routers** come in play)

Routers

- Routers
  - Intermediate nodes of IP networks which are usually connected to more than one layer two network (network interfaces and each may use different technologies
  - A router is capable of transferring IP packets between two networks, even if each uses a different layer two technology



IP Layer

- There are currently two IP versions,
  - IPV4 accounts for 80% of the network traffic
  - IPV6, released already on the 90' is gradually being deployed into the internet particularly to its backbone
- Main differences
  - IPV4 uses 32 bit addresses
  - IPV6 uses 132 bit addresses
    - This is the main driving force for the adoption of IPV6, since IPV4 addresses are almost exhausted
    - The main problem is that technology has been capable of providing multiple addresses over just one IP address!

IP network addresses

- An IP address is a unique identifier for a network. Each different layer two network needs to have a different network address.
  - Routers only retransmit IP packets between networks with different addresses (otherwise they are not different networks)
- In IPv 4 and IPv 6, the most significant bits of a node address (leftmost bits) are the network address, the remaining bits represent the address of the node within that network
- Two nodes belong to the same network if their addresses have the same network prefix, also, the remaining bits must be different because node addresses must be unique

Representation

IPv4 Address Format (Dotted Decimal Notation)

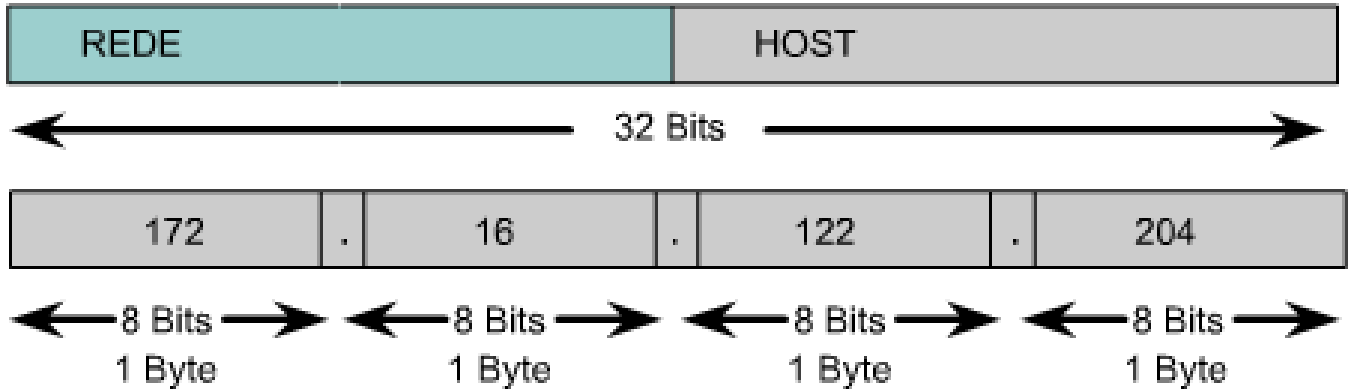
123.89.46.72

First Octet Second Octet Third Octet Fourth Octet

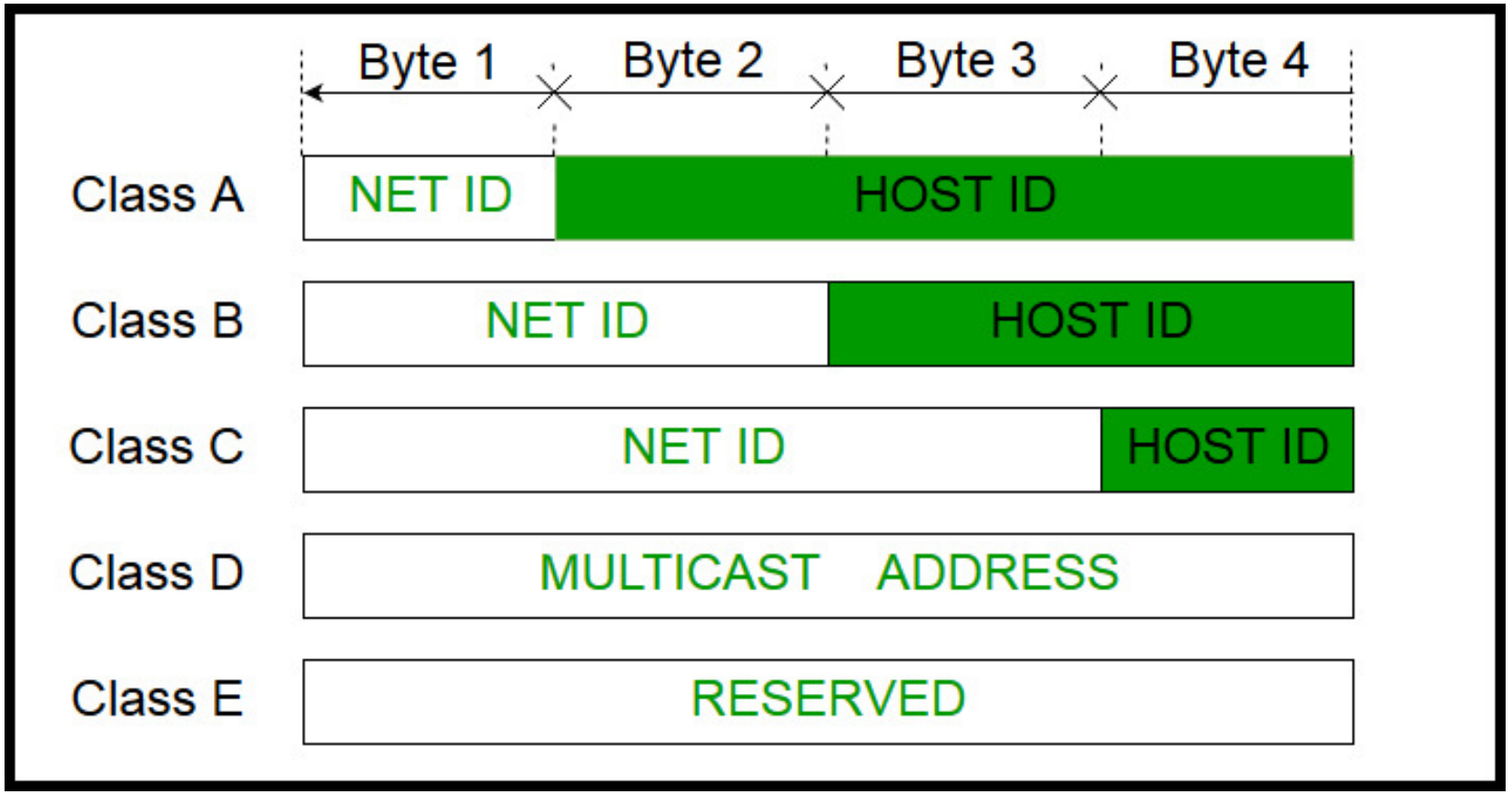
01111011.01011001.00101110.01001000

1 Byte=8 Bits

4 Bytes =32 Bits



IP Address classes



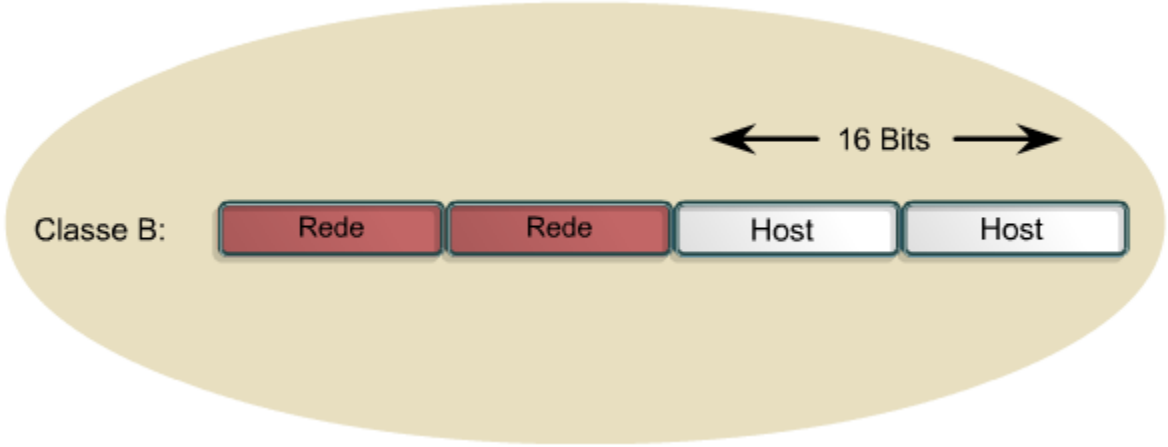
Endereçamento

- Classe A
  - **The first bit in a class A address is always 0**
    - O menor número que pode ser representado é 00000000, que também é o 0 decimal
    - O maior número que pode ser representado é 01111111, equivalente a 127 em decimal
    - **Os números 0 e 127 são reservados e não podem ser usados como endereços de rede**
    - Portanto, qualquer endereço que comece com um valor entre **1** e **126** no primeiro octeto é um endereço de classe A

Porque é reservado o 127.0.0.0 ?

Addressing

- Classe B



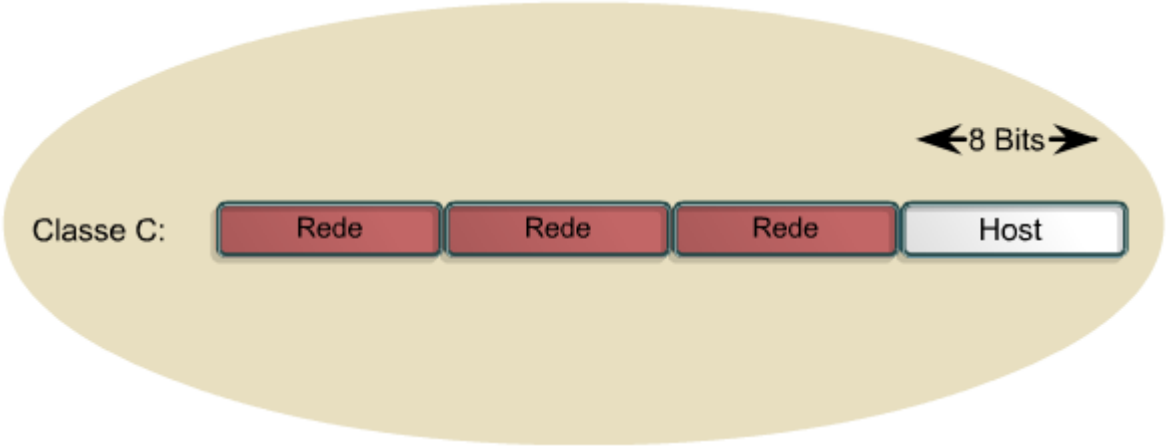
- Foi criada para suportar redes de porte médio a grande. São usados os dois primeiros octetos para indicar o endereço da rede e os outros dois octetos especificam os endereços dos hosts na rede

Endereçamento

- Classe B
  - Os dois primeiros bits de um endereço classe B são sempre 10. Os seis bits restantes podem ser preenchidos com 1s ou 0s
  - O menor número que pode ser representado por um endereço classe B é 10000000, equivalente a 128 em decimal
  - O maior número que pode ser representado é 10111111, equivalente a 191 em decimal
  - Qualquer endereço que comece com um valor no intervalo de 128 a 191 no primeiro octeto é um endereço classe B

Endereçamento

- Classe C



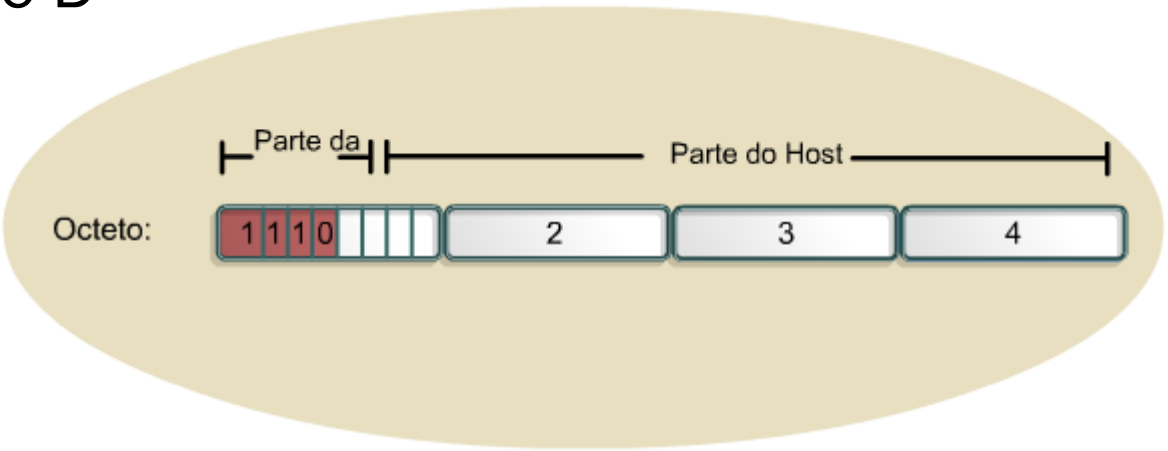
- O espaço de endereços de classe C é o mais usado. Esta classe de endereços suporta redes pequenas com um máximo de 254 *hosts*

Endereçamento

- Classe C
  - Um endereço classe C começa com os bits 110. Os cinco bits restantes podem ser preenchidos com 1s ou 0s
  - O menor número que pode ser representado é 11000000, equivalente a 192 em decimal
  - O maior número que pode ser representado é 11011111, equivalente a 223 em decimal
  - Qualquer endereço que comece com um valor no intervalo de 192 a 223 no primeiro octeto, é um endereço classe C

Endereçamento

- Classe D



- O endereço classe D foi criado para permitir multicasting num endereço IP
- Um endereço de multicast é um endereço de rede exclusivo que direcciona os pacotes destinados a esse endereço para grupos pré-definidos de endereços IP

Endereçamento

- Classe D

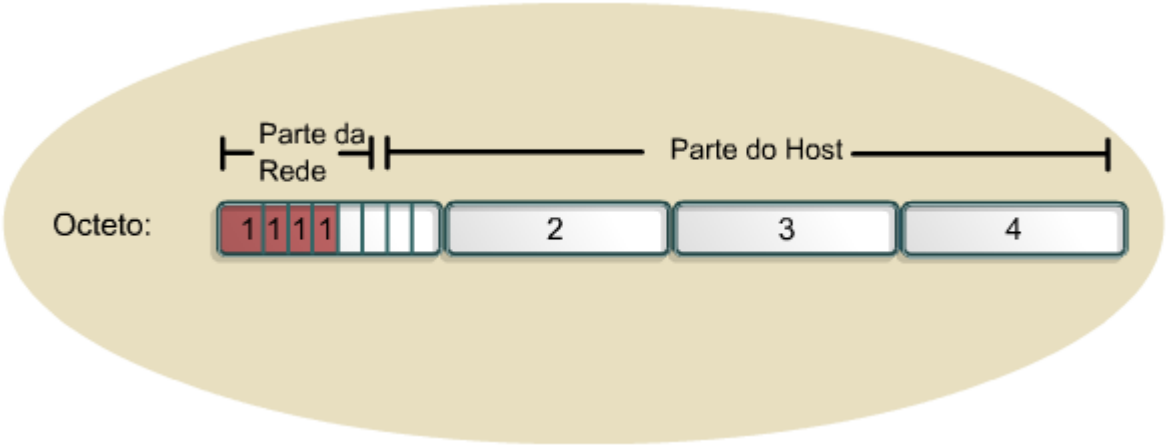
- Os primeiros quatro bits de um endereço classe D devem ser 1110. Os quatro bits restantes podem ser preenchidos com 1s ou 0s

  - O intervalo de valores no primeiro octeto dos endereços de classe D vai de 11100000 a 11101111, ou de 224 a 239 em decimal

  - Um endereço IP que comece com um valor no intervalo de 224 a 239 no primeiro octeto é um endereço classe D

Endereçamento

- Classe E



- Os endereços desta classe são reservados pela Internet Engineering Task Force (IETF) para investigação

Endereçamento

- Classe E

- Os primeiros quatro bits de um endereço classe E são sempre definidos como 1s. Os quatro bits restantes podem ser preenchidos com 1s ou 0s

  - O intervalo de valores no primeiro octeto dos endereços de classe E vai de 11110000 a 11111111, ou seja de 240 a 255 em decimal

Address Classes

Classe de endereços IP	Intervalo de endereços IP (Valor Decimal do Primeiro Octeto)
Classe A	1-126 (00000001-01111110) *
Classe B	128-191 (10000000-10111111)
Classe C	192-223 (11000000-11011111)
Classe D	224-239 (11100000-11101111)
Classe E	240-255 (11110000-11111111)

Endereçamento

Classe de Endereço	Número de Redes	Número de Hosts por Rede
A	126 *	16,777,216
B	16, 384	65,535
C	2,097,152	254
D (Multicast)	N/A	N/A

Classe de Endereço IP	Bits de Ordem Superior	Intervalo de Endereços do Primeiro Octeto	Número de Bits no Endereço de Rede
Classe A	0	0 - 127 *	8
Classe B	10	128 - 191	16
Classe C	110	192 - 223	24
Classe D	1110	224 - 239	28

4

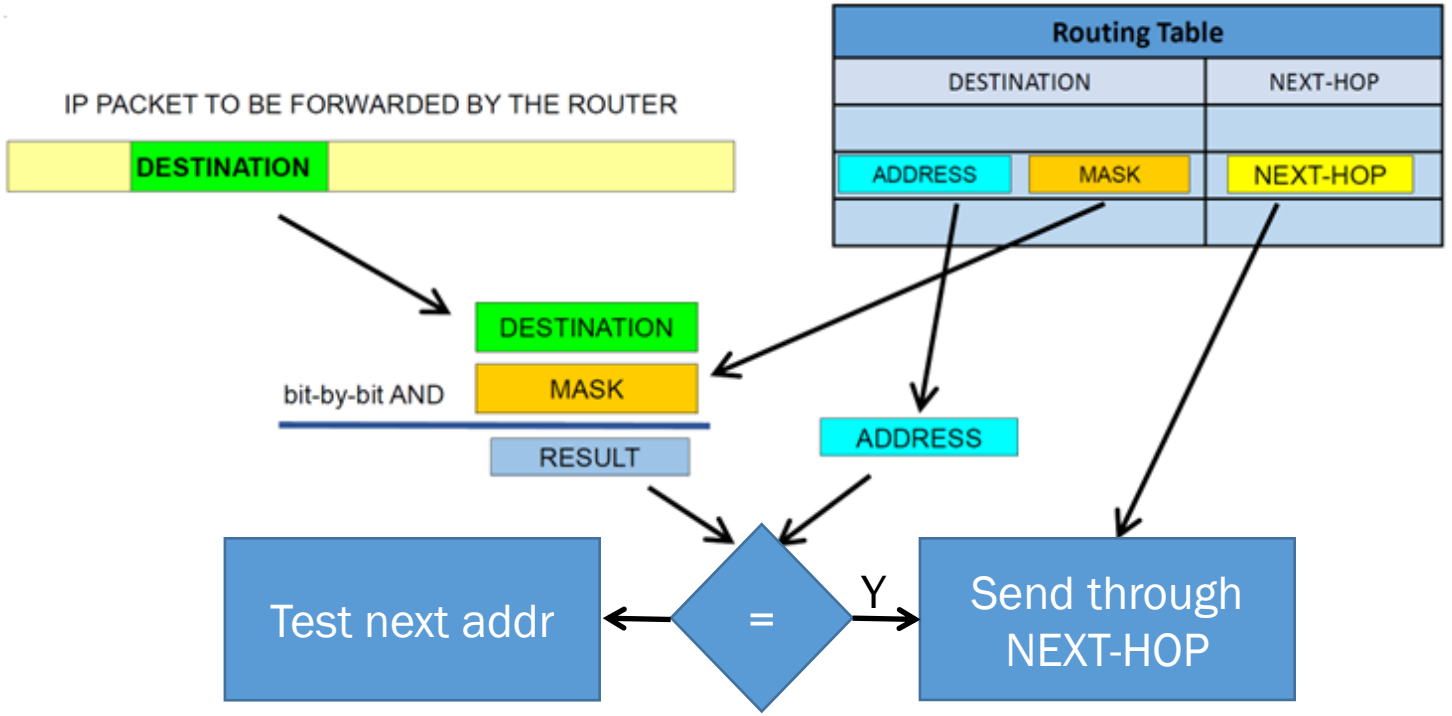
Routing

- Switches and Routers
  - These nodes receive packets intended to other nodes, forwarding those packets to other nodes and ultimately to the destination node
  - Switches
    - Layer two
  - Routers or gateways
    - Layer three
    - Several possible interfaces
    - Different network protocols

Routing

- Each router must decide for each packet it receives where should it be sent to (what is the next hop).
- Alternative paths may exist, and less optimal decisions may be taken
- Routers interact only with other neighboring routers, those which are the possible next hops A remote router can never be a next hop
- Routing decision are supported by routing tables containing the destination address (or group of addresses) and the next hop for that address

Routing table



Routing tables: creation and update

- Routing tables can be manually created - **static routing**
- But we can also make routers talk to each other to automatically build routing tables - **dynamic routing**
- Dynamic routing not only ensures the initial building of tables but also keeps them updated.
  - When there is a change in the infrastructure, that change is automatically reflected in the routing tables.
- The process of reflecting in the routing tables a change in the infrastructure is called convergence The time it takes, the **convergence time**
- Taking advantage of alternative paths (**fault tolerance and load balancing**) requires the use of dynamic routing. This comes obvious, as, with static routing, packets will always follow the same exact path

TCP and UPD protocols

Client/Server

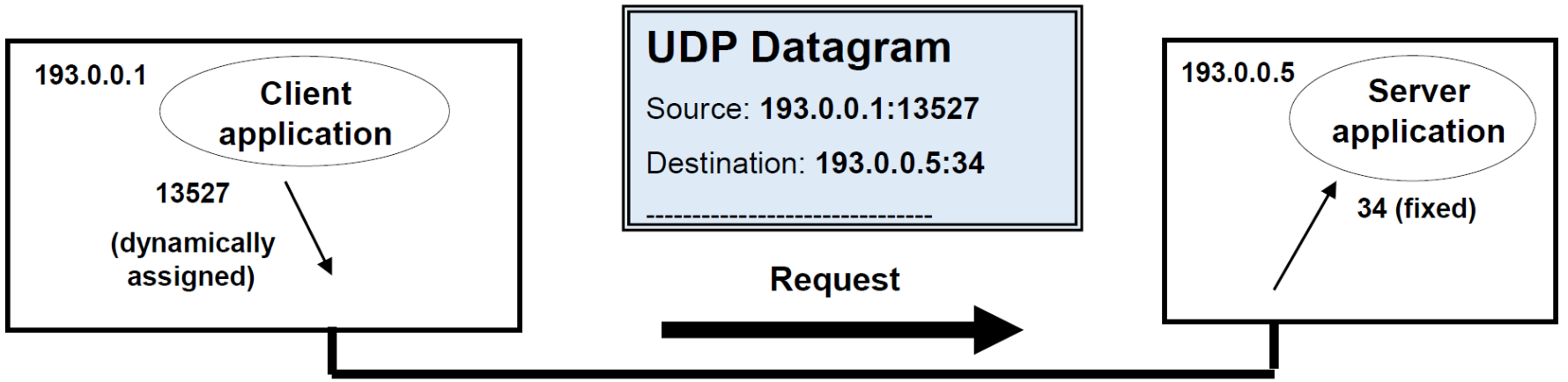
- Most network applications use the client/server model

- The **Server** has a fixed local port number (pre agreed with the client) and **waits for requests from clients**.

- The **Client** can use any local port number (which can be dynamically assigned) on his side.
  - **It MUST initiate communications** with the Server, by knowing its IP address and port number

User Datagram Protocol (UDP)

- Source and destination applications for each UDP datagram are identified by 16 bit numbers - the **port** numbers plus the network address



UDP characteristics

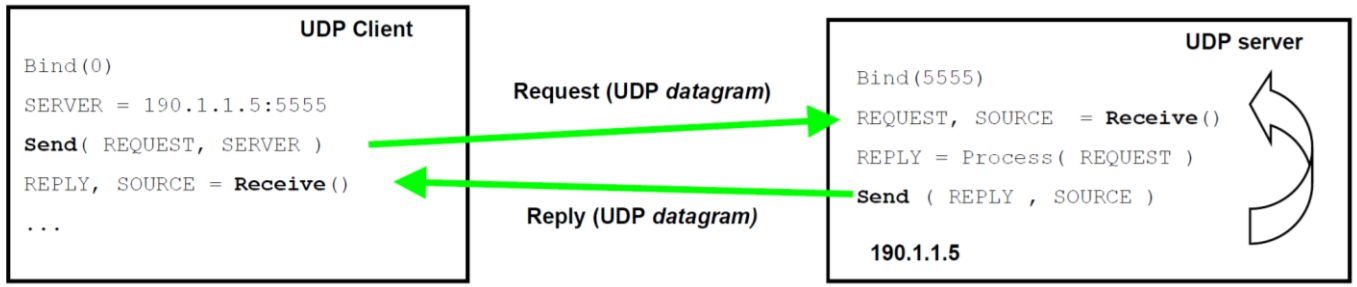
- UDP is not connection oriented (it is **connection less**)
  - Does not offer any kind of delivery guarantee
    - consequently frames can be lost and the senders are note able to know if frames were delivered
  - Does not guarantee the order of frames in consecutive transmissions
  - Does not offer any kind of flow control
  - Each datagram is handled individually
    - for each datagram to be sent, a destination IP address and a destination port number must be provided by the sender
  - For a sent datagram to be received, on the specified destination IP address, an UDP application should be running and listening, on the specified destination port number
  - UDP supports the broadcast of frames

UDP characteristics

- Sending and receiving operations are managed through FIFO queues. From the instant a local port number is assigned to the UDP socked, received datagrams are queued.
- UDP datagrams are sent by calling a system call that requires a data block, the data block size and the destination address (IP address and the destination UDP port number)
- UDP datagrams are received by calling a system call that returns a data block, the data block size and the source address (IP address and the source UDP port number).
  - If the socket's input queue is empty, the **operation usually blocks** the process or thread until a datagram is received

UDP characteristics - delivery

- There is no warranty a sent datagram will ever reach the destination, neither any feedback to the sender concerning the delivery success
- **Application protocols must implement these features when required**



- The main problem is that the client can block forever if no reply is received from the server

UDP characteristics – delivery

- Solutions for guaranteed delivery problem can be:
  - Non blocking sockets:
    - Operations on the socket that can't be executed immediately (would block the thread/process), will instead return immediately with an error
  - Sockets with timeout:
    - If the operation over the socket takes longer than the timeout, the operation will abort with an error
  - Threads or processes:
    - Create a separate thread or process to perform the blocking operation, thus, the main thread or process is not blocked. The created thread or process can also be terminated after a period of time.
  - Sockets monitoring:
    - Some languages provide functions capable of detecting when a socket is ready for reading The **select** function in C language allows this and also the setting of a timeout

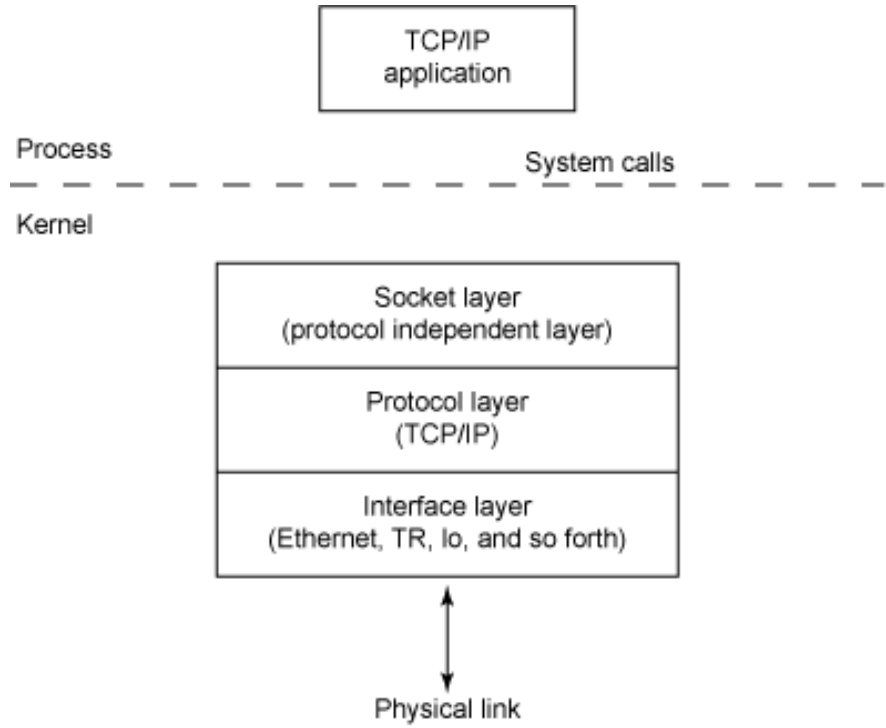
UDP characteristics – datagram size

- An IPv 4 datagram can be up to 65535 bytes long, but transmitting such amount of data depends on the underlying MAC protocol and on the settings/implementation of the TCP/IP protocol
- The safe way to ensure the excessive size of a UDP datagram will not compromise its delivery, is **avoiding its content length to be above 512 bytes** (RFC 791). Solutions:
  - Use TCP
  - Split the data into smaller datagrams
    - As minimum procedures, the receiver of a set of datagrams must be informed of the total number of datagrams, also each datagram requires a sequence number
    - How to handle error recovery?

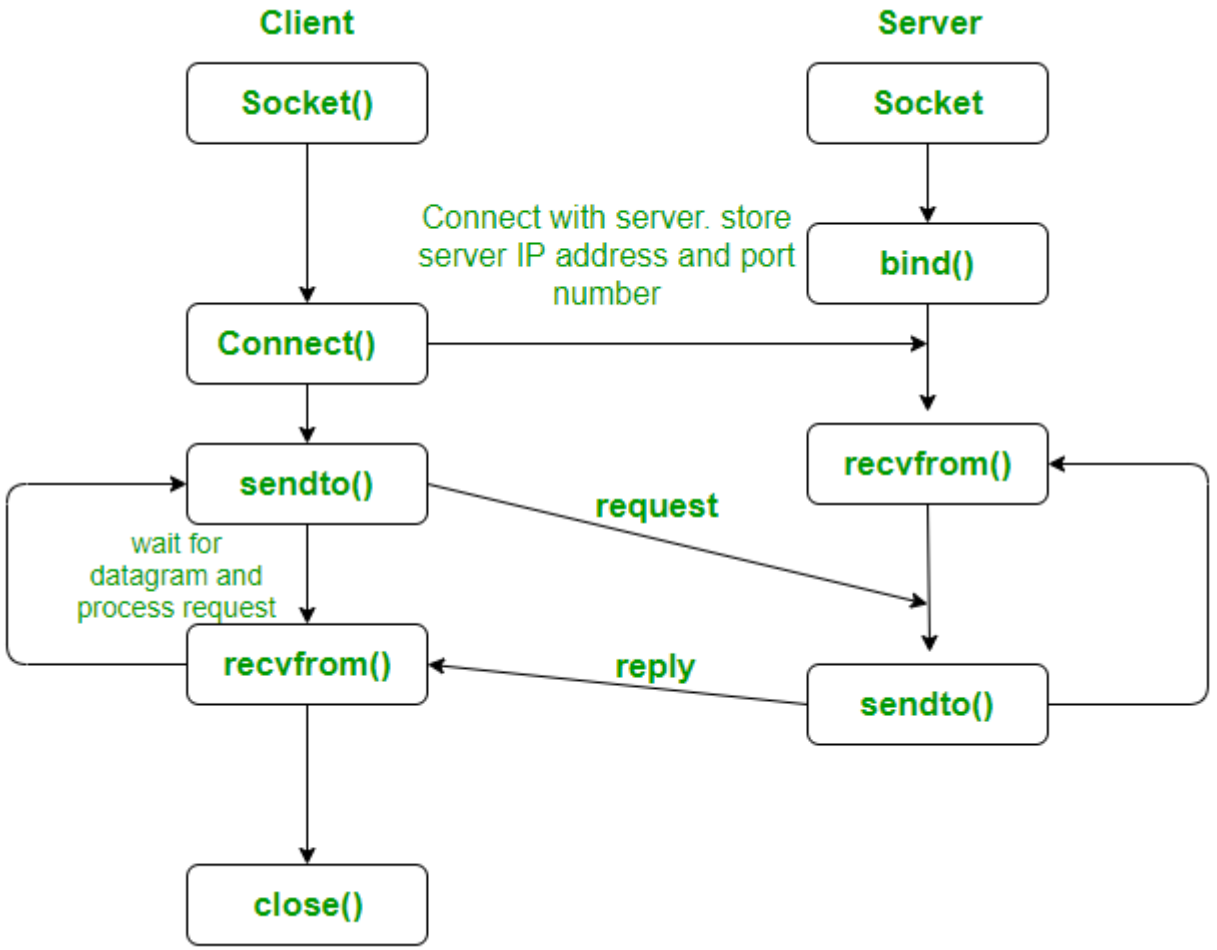

isep
Instituto Superior de
Engenharia do Porto
P.PORTO

The Socket layer

- Any TCP system call that is made is received by the socket layer.
- The socket layer validates the correctness of the parameters passed by the TCP application.
- This is a protocol-independent layer because the protocol is not yet hooked onto the call.



Establishing an UDP communication



Functions

- `int socket(int domain, int type, int protocol)`
  - Creates an unbound socket in the specified domain. Returns socket file descriptor.
  - `domain` – Specifies the communication protocol (**AF_INET** for IPv4/ AF_INET6 for IPv6)
  - `type` – Type of socket to be created (SOCK_STREAM for TCP / SOCK_DGRAM for UDP)
  - `protocol` – Protocol to be used by the socket. 0 means use default protocol for the address family.

Functions - bind

- `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)`
  - Assigns address/port to an unbound socket.
  - `sockfd` – File descriptor of a socket to be bonded, obtained on the return from the socket function
  - `addr` – Structure in which address to be bound to is specified
  - `addrlen` – Size of addr structure

Functions - sendTo

- `ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen)`
  - Send a message on the socket
  - `sockfd` – File descriptor of the socket
  - `buf` – Application buffer containing the data to be sent
  - `len` – Length of buf
  - `flags` – Bitwise OR of flags to modify socket behaviour
  - `dest_addr` – Structure containing the address of the destination
  - `addrlen` – Size of `dest_addr` structure

Functions - recvFrom

- `ssize_t recvfrom(int sockfd, void *restrict buffer, size_t length, int flags, struct sockaddr *restrict address, socklen_t *restrict address_len);`
  - Receives a message from a socket. Normally used with connectionless-mode sockets because it permits the application to retrieve the source address of received data.
  - Upon successful completion, recvfrom() shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, `recvfrom()` shall return 0. Otherwise, the function shall return -1 and set errno to indicate the error.
  - `Sockfd`: Specifies the socket file descriptor.
  - `Buffer`: Points to the buffer where the message should be stored.
  - `Length`: Specifies the length in bytes of the buffer pointed to by the buffer argument.

Functions – sendTo (cont)

- `Flags`: Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:
  - MSG_PEEK: Peeks at an incoming message. The data is treated as unread and the next `recvfrom()` or similar function shall still return this data.
  - MSG_OOB: Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
  - **MSG_WAITALL**: On a SOCK_STREAM socket requests that the function blocks until the full amount of data can be returned. The function may return a smaller amount of data if the socket is a message-based socket, if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket.
- `Address`: A null pointer, or points to a `sockaddr` structure in which the sending address is to be stored. The length and format of the address depend on the address family of the socket.
- `address_len`: Specifies the length of the sockaddr structure pointed to by the address argument.

UCP Server example (1)

```
// 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
```

UCP Server example (2)

```
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);
    }
```

UCP Server example (3)

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

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

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

UCP Server example (4)

```
int len, n;
len = sizeof(cliaddr);  //len is value/result
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;
}
```

UCP Client example (1)

```
// 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
```

UCP Client example (2)

```
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);
    }
```

UCP Client example (3)

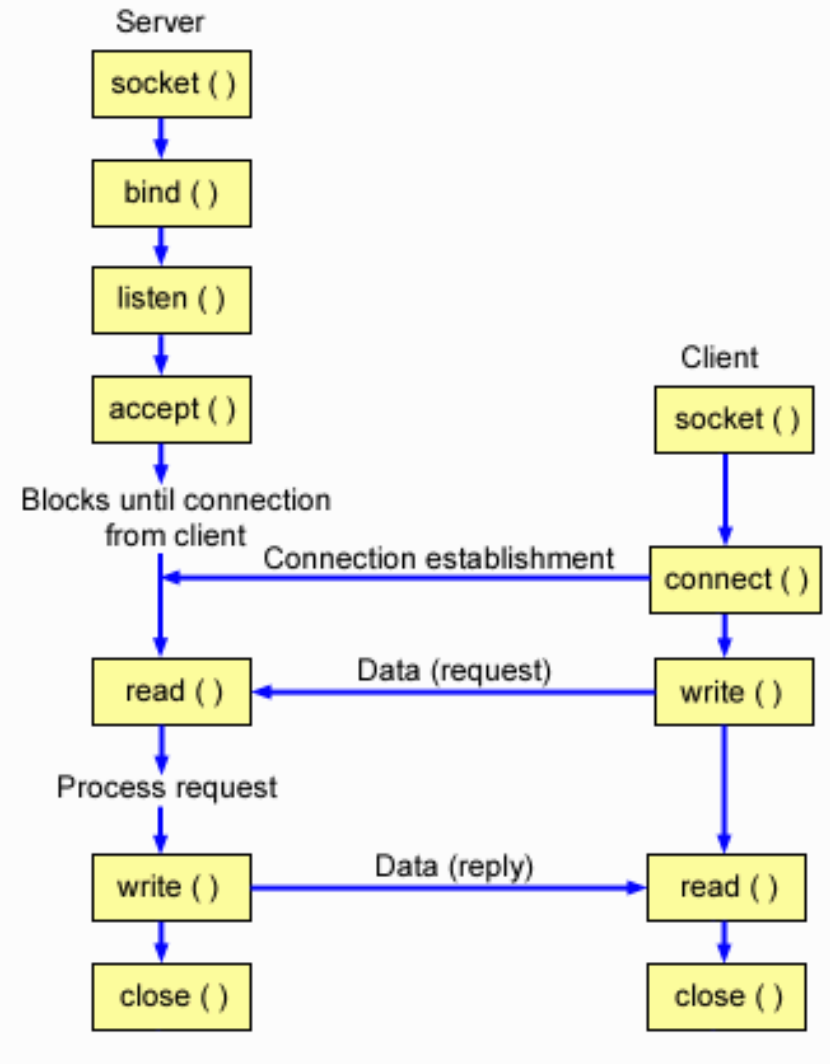
```
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = INADDR_ANY;
    int n, len;

    sendto(sockfd, (const char *)hello, strlen(hello),
        MSG_CONFIRM, (const struct sockaddr *) &servaddr, sizeof(servaddr));
    n = recvfrom(sockfd, (char *)buffer, MAXLINE, MSG_WAITALL, (struct sockaddr *) &servaddr,
&len);
    buffer[n] = '\0';
    printf("Server : %s\n", buffer);
    close(sockfd);
    return 0;
}
```

Transmission Control Protocol (TCP)

- TCP protocol provides a significantly higher quality of service (QoS) when compared to UDP.
- Creates logical bidirectional communication channels between applications located in different network nodes
  - These logical channels, commonly referred to as TCP connections, provide guarantees on data delivery and data sequence
  - Each TCP connection is for exclusive use by the two applications that created it
- Sending and receiving data through a TCP connection is flux oriented
  - i.e. there is no data block concept like in UDP, **all data is sent and received on a byte by byte continuous flow**

TCP Connection Establishment



TCP Connection Establishment

- Socket
  - A socket is one endpoint of a two-way communication link between two programs running on the network (or the same computer). A socket is usually bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.
- Bind
  - Both the client and the server 'bind' to a particular port on their machines and to a socket. The binding procedure on the client is entirely optional. The bind system call requires the address family, the port number and the IP address.
  - **The server application is then available to receive connection requests on the specified TCP port number**

TCP Connection Establishment

- Listen
  - **Marks the socket on its arguments as a passive socket,** that is, as a socket that will be used to accept incoming connection requests (using `accept()`)
  - The connect function can now be called on the client since the server will be waiting for a connect call
- Accept
  - The request generated by a connect call is placed in an operating system buffer, waiting to be handed over to the application, which only happens by calling the accept function. This call is the mechanism by which the server program receives those requests that have been accepted by the operating system.
  - **A new socket is created just to handle this connection, the original one remains open, in order to accept other calls**

TCP Connection Establishment

- Connect
  - Initiates a connection on a socket with a server application on a specific address/port

- Data can now be transferred using simple read and write functions, like:
  - `int recv(int skfd, void *buf, int buflen, int flags);`
  - `int send(int skfd, void *buf, int buflen, int flags);`

Functions - listen

- `int listen(int socket, int backlog);`
  - The `listen()` function shall mark a TCP (connection-mode) socket, specified by the socket argument, as accepting connections.
  - `Socket`: the associated socket
  - The `backlog` argument defines an upper limit on the length of the queue of pending connections, either globally or per accepting socket.
    - If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection succeeds

Function - accept

- `int accept(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);`
  - extracts the first connection on the queue of pending connections, creates a new socket with the same socket type protocol and address family as the specified socket, and allocates a new file descriptor for that socket.
  - `socket`: Specifies a socket that was created with `socket()`, has been bound to an address with `bind()`, and has issued a successful call to `listen()`.
  - `address`: Either a null pointer, or a pointer to a `sockaddr` structure where the address of the connecting socket shall be returned.
  - `address_len`: length of the supplied `sockaddr` structure

Function - connect

- `int connect(int socket, const struct sockaddr *address, socklen_t address_len);`
  - Attempts to make a connection on a socket.
  - `Socket`: Specifies the file descriptor associated with the socket.
  - `Address`: Points to a `sockaddr` structure containing the peer address. The length and format of the address depend on the address family of the socket.
  - `address_len`: Specifies the length of the `sockaddr` structure pointed to by the address argument.

TCP server example (2)

```
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#define MAX 80
#define PORT 8080
#define SA struct sockaddr
```

TCP server example (2)

```
void func(int connfd) // Function to chat between client and server. {
    char buff[MAX]; int n;
    for (;;) {
        bzero(buff, MAX);
        read(connfd, buff, sizeof(buff));
        printf("From client: %s\t To client : ", buff); bzero(buff, MAX);
        n = 0;
        while ((buff[n++] = getchar()) != '\n');
        write(connfd, buff, sizeof(buff));
        if (strncmp("exit", buff, 4) == 0) { printf("Server Exit...\n");
             break;        }
    }
}
```

TCP server example (3)

```
int main(){
    int sockfd, connfd, len;
    struct sockaddr_in servaddr, cli;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
      printf("socket creation failed...\n");
      exit(0);
    }
    else  printf("Socket successfully created..\n");
    bzero(&servaddr, sizeof(servaddr));
```

TCP server example (4)

```
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(PORT);
if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
    printf("socket bind failed...\n"); exit(0);    }
if ((listen(sockfd, 5)) != 0) {
    printf("Listen failed...\n"); exit(0);    }
else
    printf("Server listening..\n");
```

TCP server example (5)

```
len = sizeof(cli);
// Accept the data packet from client and verification
connfd = accept(sockfd, (SA*)&cli, &len);
if (connfd < 0) {
    printf("server accept failed...\n"); exit(0);
}
else
    printf("server accept the client...\n");

// Function for chatting between client and server
func(connfd);
```

TCP – Client (1)

```
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#define MAX 80
#define PORT 8080
#define SA struct sockaddr
```

TCP – Client (2)

```
void func(int sockfd) {
    char buff[MAX]; int n;
    for (;;) {
        bzero(buff, sizeof(buff));
        printf("Enter the string : ");
        n = 0;
        while ((buff[n++] = getchar()) != '\n');
        write(sockfd, buff, sizeof(buff));
        bzero(buff, sizeof(buff));
        read(sockfd, buff, sizeof(buff));
        printf("From Server : %s", buff);
        if ((strncmp(buff, "exit", 4)) == 0) {
            printf("Client Exit...\n"); break;
        }
```

TCP – Client (3)

```
#int main() {
    int sockfd, connfd; struct sockaddr_in servaddr, cli;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        printf("socket creation failed...\n"); exit(0);
    }
    bzero(&servaddr, sizeof(servaddr));

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servaddr.sin_port = htons(PORT);
```

TCP – Client (4)

```
    if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr)) != 0) {
        printf("connection with the server failed...\n");
        exit(0);
    }

    // function for chat
    func(sockfd);

    // close the socket
    close(sockfd);
}
```

- Fim aula 1

Reliable Transmission of packets

- https://book.systemsapproach.org/direct/reliable.html?highlight=sliding%20window
- https://web.cs.wpi.edu/~rek/Undergrad_Nets/B06/B06.html
- https://booksite.elsevier.com/9780123850591/lec.php

Reliable Transmission

- The CRC bytes in communication frames are used to detect errors
- Some other error codes are powerfull enough to even correct errors, but the overhead is typically high
- Corrupt frames must be discarded
- A link-level protocol that wants to deliver frames reliably must recover from these discarded frames.
- This is accomplished using a combination of two fundamental mechanisms
  - **Acknowledgements and Timeouts**

Reliable Transmission

- An acknowledgement (ACK for short) is a small control frame that a protocol sends back to its peer saying that it has received the earlier frame.
- A control frame is a frame with header only (no data).
- The receipt of an acknowledgement indicates to the sender of the original frame that its frame was successfully delivered.

Reliable Transmission

- If the sender does not receive an acknowledgment after a reasonable amount of time – a **timeout**, then it retransmits the original frame.

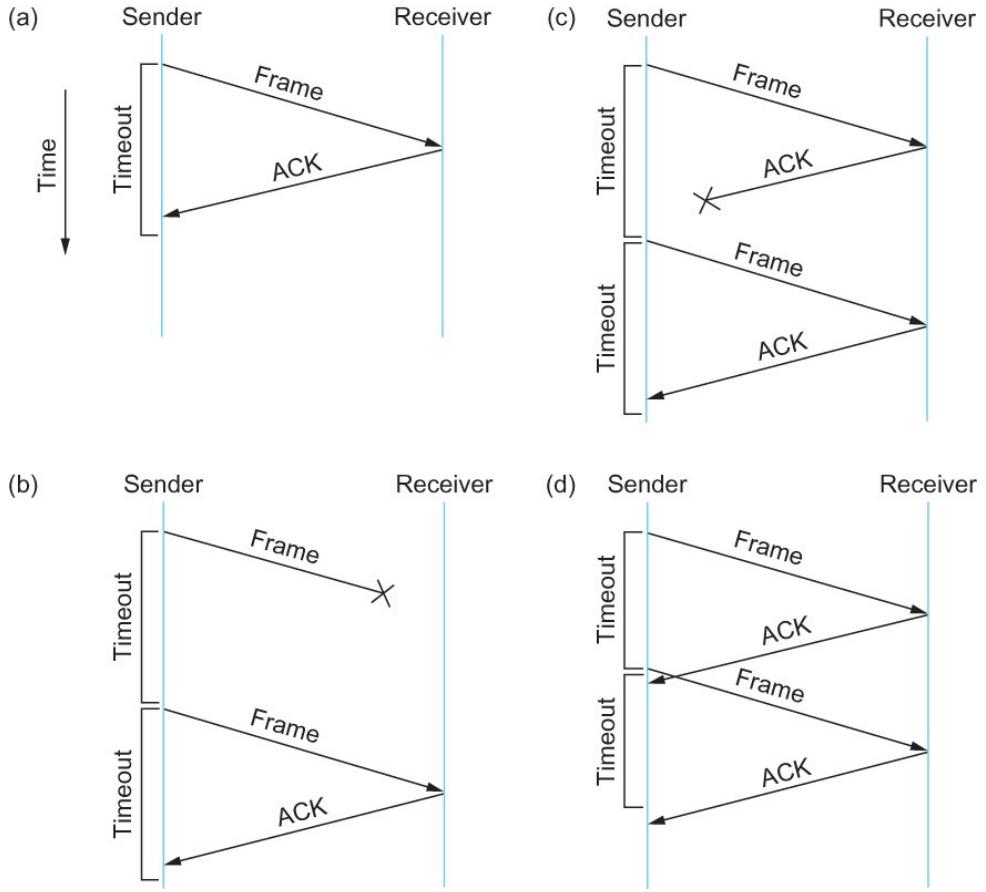
- The general strategy of using acknowledgements and timeouts to implement reliable delivery is sometimes called **Automatic Repeat reQuest (ARQ)**.

Reliable Transmission

- Stop and Wait Protocol
  - After transmitting one frame, the sender waits for an acknowledgement before transmitting the next frame.
  - If the acknowledgement does not arrive after a timeout, the sender retransmits the original frame

Reliable Transmission

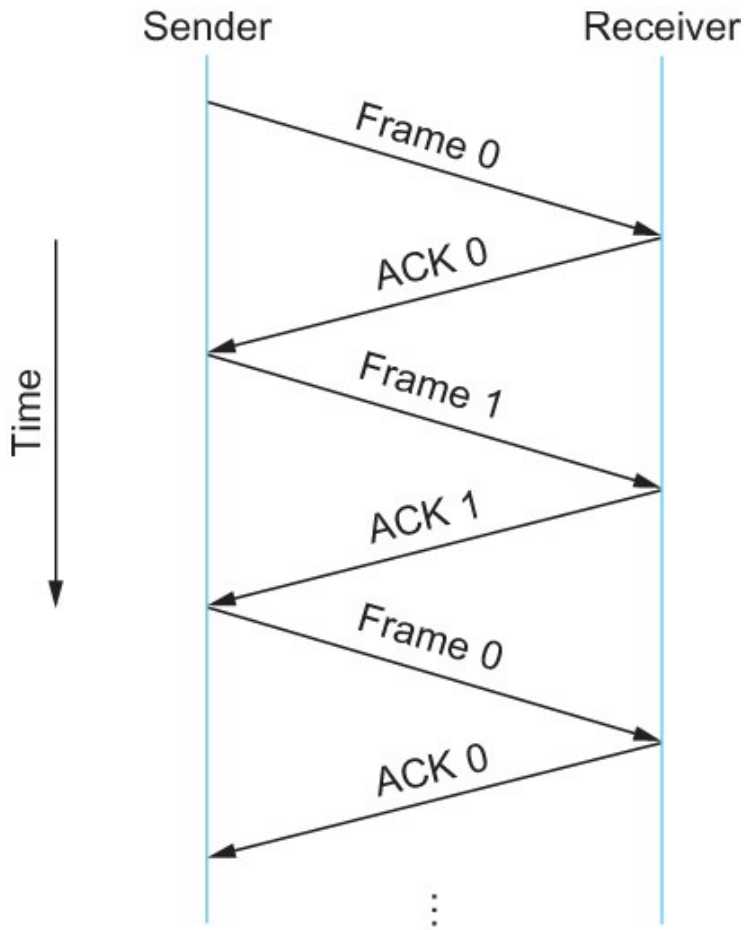
- Stop and Wait



Reliable Transmission

- Stop and Wait protocol
  - If the acknowledgment is lost or delayed in arriving
    - The sender times out and retransmits the original frame, but the receiver will think that it is the next frame since it has correctly received and acknowledged the first frame
    - As a result, **duplicate copies of frames will be delivered**
  - How to solve this?
    - Use 1 bit sequence number (0 or 1)
    - When the sender retransmits frame 0, the receiver can determine that it is seeing a second copy of frame 0 rather than the first copy of frame 1 and therefore can ignore it (the receiver still acknowledges it, in case the first acknowledgement was lost)

Reliable Transmission



Reliable Transmission

- Stop and Wait protocol: Problems
  - The sender has only one outstanding frame on the link at a time! This may be far below the link's capacity. E.g. when the link has several hops
- Consider a 1.5 Mbps link with a 45 ms Rount Trip Time (RTT)
  - The link has a delay × bandwidth product of 67.5 Kb or approximately 8 KB
  - Since the sender can send only one frame per RTT and assuming a frame size of 1 KB
  - Maximum Sending rate
    - Bits per frame ÷ Time per frame = 1024 × 8 ÷ 0.045 = 182 Kbps

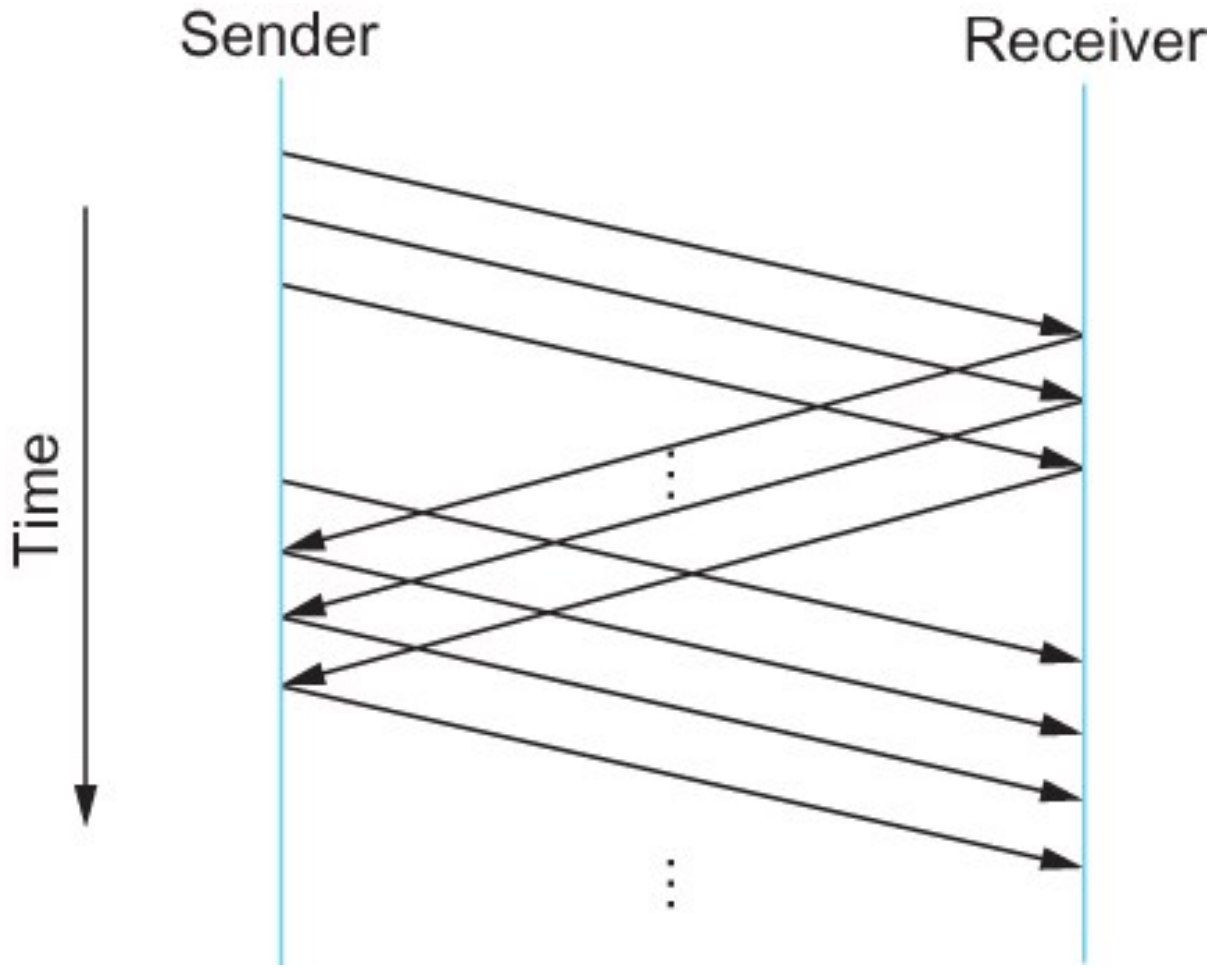
      Or **about one-eighth of the link's capacity**

  - **To use the link fully, then the sender must transmit up to eight frames before having to wait for an acknowledgement – waste of bandwidth**

Reliable Transmission

- Solution:

Sliding window protocol



Sliding window protocol

- Sender assigns a sequence number denoted as SeqNum to each frame.

- Sender maintains three variables
  - Sending Window Size (SWS)
    - Upper bound on the number of outstanding (unacknowledged) frames that the sender can transmit
  - Last Acknowledgement Received (LAR)
    - Sequence number of the last acknowledgement received
  - Last Frame Sent (LFS)
    - Sequence number of the last frame sent

- Por no Quadro!!!

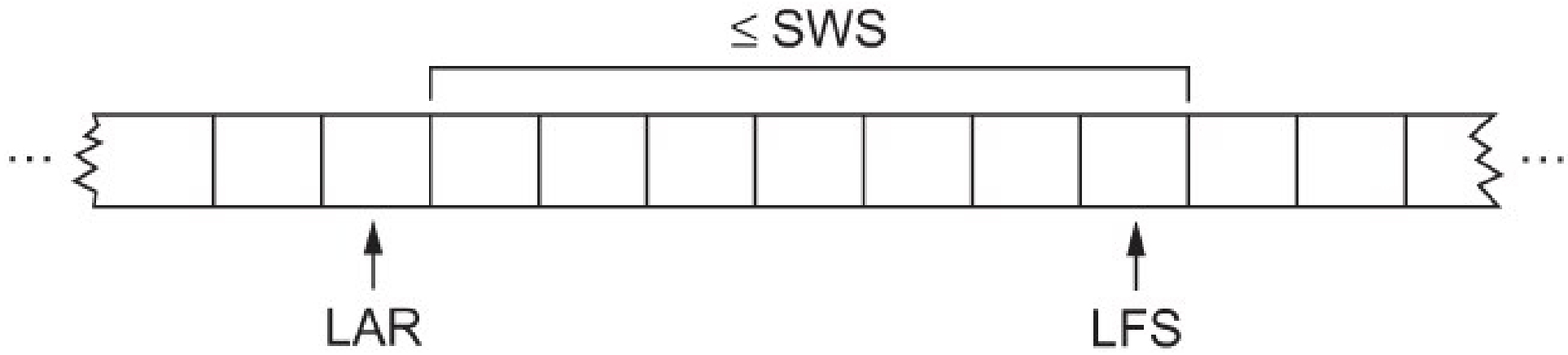
Sliding window protocol

- When an acknowledgement arrives
  - the sender moves LAR to the right, thereby allowing the sender to transmit another frame
- The sender associates a timer with each frame it transmits
  - It retransmits the frame if the timer expires before the ACK is received
- Note that the sender must buffer up to SWS frames
  - WHY?

Sliding window protocol

- Sender also maintains the following condition

$$LFS - LAR \leq SWS$$



Last Acknowledgement Received (LAR)

Last Frame Sent (LFS)

Sliding window protocol

- Receiver maintains three variables
  - Receiving Window Size (RWS)
    - Upper bound on the number of out-of-order frames that the receiver is willing to accept
  - Largest Acceptable Frame (LAF)
    - Sequence number of the largest acceptable frame
  - Last Frame Received (LFR)
    - Sequence number of the last frame received

Sliding window protocol

- When a frame with sequence number SeqNum arrives, what does the **receiver** do?
  - If SeqNum ≤ LFR or SeqNum > LAF
    - Discard it (the frame is outside the receiver window)
  - If LFR < SeqNum ≤ LAF
    - Accept it
    - Now the receiver needs to decide whether or not to send an ACK

Sliding window protocol

- Let SeqNumToAck
  - Denotes the largest sequence number not yet acknowledged, such that all frames with sequence number less than or equal to SeqNumToAck have been received
- The receiver acknowledges the receipt of SeqNumToAck even if high-numbered packets have been received
  - This acknowledgement is said to be cumulative.
- The receiver then sets
  - LFR = SeqNumToAck and adjusts
  - LAF = LFR + RWS

Sliding window protocol

For example, suppose LFR = 5 and RWS = 4 (i.e. the last ACK that the receiver sent was for seq. no. 5)

$\Rightarrow$LAF = 9

If frames 7 and 8 arrive, they will be buffered because they are within the receiver window, but no ACK will be sent since frame 6 did not arrive yet

Frames 7 and 8 are out of order

Frame 6 arrives (it is late because it was lost the first time and had to be retransmitted)

Now Receiver Acknowledges Frame 8 and LFR goes to 8 and LAF to 12

Sliding window protocol

- When a timeout occurs, the amount of data in transit decreases
  - Since the sender is unable to advance its window
- When a packet loss occurs, this scheme is no longer keeping the pipe full
  - The longer it takes to notice that a packet loss has occurred, the more severe the problem becomes
- How to improve this
  - Negative Acknowledgement (NAK)
  - Additional Acknowledgement
  - Selective Acknowledgement

Sliding window protocol

- Negative Acknowledgement (NAK)
  - Receiver sends NAK for frame 6 when frame 7 arrive (in the previous example)
    - However, this is unnecessary since sender's timeout mechanism will be sufficient to catch the situation
- Additional Acknowledgement
  - Receiver sends additional ACK for frame 5 when frame 7 arrives
    - Sender uses duplicate ACK as a clue for frame loss
- Selective Acknowledgement
  - Receiver will acknowledge exactly those frames it has received, rather than the highest number frames
    - Receiver will acknowledge frames 7 and 8
    - Sender knows frame 6 is lost
    - Sender can keep the pipe full (additional complexity)


isep
Instituto Superior de
Engenharia do Porto
P.PORTO
Fim aula

Sliding window protocol

How to select the window size

- SWS is easy to compute
  - Delay × Bandwidth
- RWS can be anything
  - Two common setting
    - RWS = 1

      No buffer at the receiver for frames that arrive out of order
    - RWS = SWS

      The receiver can buffer all frames that a sender can transmit

It does not make any sense to keep RWS > SWS

WHY?

Sliding window protocol

- Finite Sequence Number
  - Frame sequence number is specified in the header field
    - Finite size
      - 3 bit: eight possible sequence number: 0, 1, 2, 3, 4, 5, 6, 7
    - It is necessary to wrap around

Sliding window protocol

- How to distinguish between different incarnations of the same sequence number?
  - Number of possible sequence number must be larger than the number of outstanding frames allowed
    - Stop and Wait: One outstanding frame
      - 2 distinct sequence number (0 and 1)
    - Let MaxSeqNum be the number of available sequence numbers
    - SWS + 1 ≤ MaxSeqNum
      - Is this sufficient?

Sliding window protocol

SWS + 1 ≤ MaxSeqNum

- Is this sufficient?
- Depends on RWS
- If RWS = 1, then sufficient
- If RWS = SWS, then not good enough

- For example, we have eight sequence numbers

  0, 1, 2, 3, 4, 5, 6, 7

  RWS = SWS = 7

  Sender sends 0, 1, …, 6

  Receiver receives 0, 1, … ,6

  Receiver acknowledges 0, 1, …, 6

  ACK (0, 1, …, 6) are lost

  Sender retransmits 0, 1, …, 6

  Receiver is expecting 7, 0, …., 5

Sliding window protocol

To avoid this,

If RWS = SWS

SWS < (MaxSeqNum + 1)/2

Sliding window protocol

- Serves three different roles
  - Reliable
  - Preserve the order
    - Each frame has a sequence number
    - The receiver makes sure that it does not pass a frame up to the next higher-level protocol until it has already passed up all frames with a smaller sequence number
  - Frame control
    - Receiver is able to *throttle* the sender, i.e.:
      - Keeps the sender from overrunning the receiver
      - From transmitting more data than the receiver is able to process

TCP/IP Congestion control

- The congestion problem
  - Congestion control was introduced in the late 1980s by Van Jacobson, roughly eight years after the TCP/IP protocol stack had become operational.
  - Immediately preceding this time, the Internet was suffering from congestion collapses—hosts would send their packets into the Internet as fast as the advertised window would allow, congestion would occur at some router (causing packets to be dropped), and the hosts would time out and retransmit their packets, resulting in even more congestion.

TCP/IP Congestion control

- Essential strategy:
  - The TCP host sends packets into the network without a reservation and then the host reacts to observable events
- Originally TCP assumed FIFO queuing

- Basic idea:
  - each source determines how much capacity is available to a given flow in the network.
  - ACKs are used to ‘pace’ the transmission of packets such that TCP is “self-clocking”.

Additive Increase / Multiplicative Decrease

- CongestionWindow (`cwnd`) is a variable held by the TCP source for each connection

```
MaxWindow :: min (CongestionWindow , AdvertisedWindow)
EffectiveWindow = MaxWindow – (LastByteSent -LastByteAcked)
```

- `cwnd` is set based on the perceived level of congestion. The Host receives implicit (packet drop) or explicit (packet mark) indications of internal congestion.

Additive Increase

- Additive Increase is a reaction to perceived available capacity.
- **Linear Increase basic idea:**
  - For each “cwnd’s worth” of packets sent, increase cwnd by 1 packet
- In practice, **cwnd is incremented fractionally for each arriving ACK.**

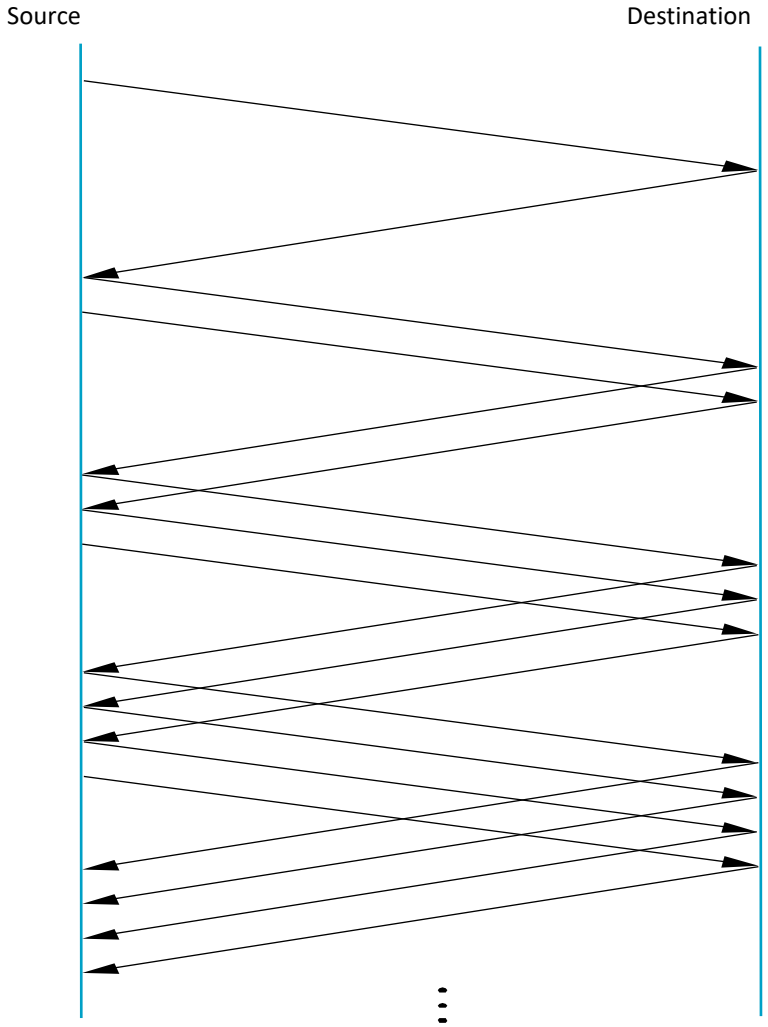
  increment = MSS x (MSS /cwnd)

  cwnd = cwnd + increment

- Maximum Segment Size (MSS)

Additive Increase

- Strategy: add one packet every RTT



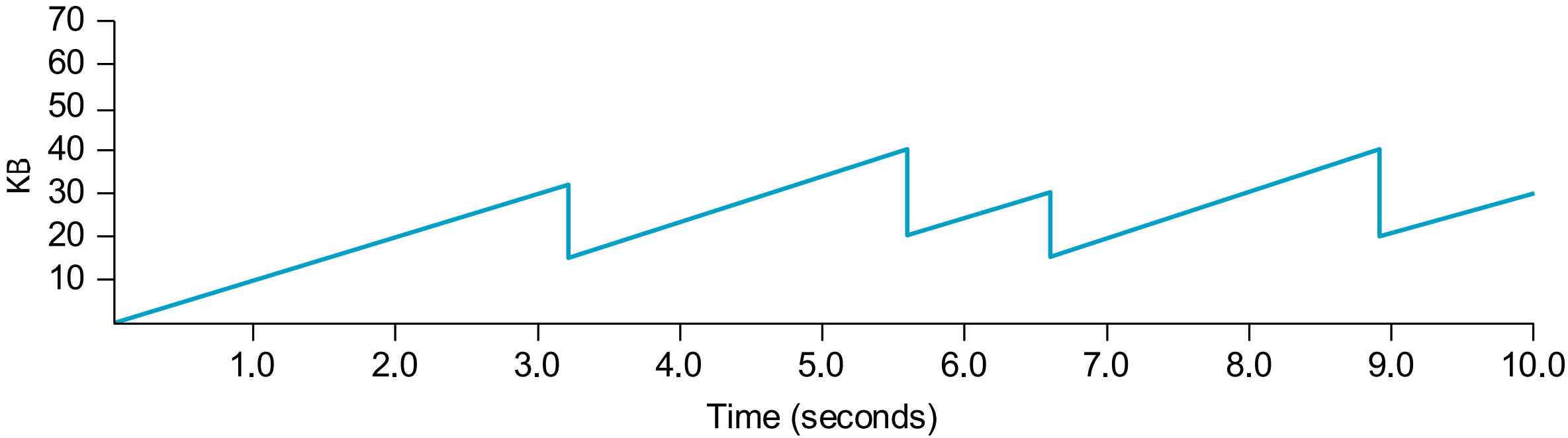
Multiplicative Decrease

- The key assumption is that a dropped packet and the resultant timeout are due to congestion at a router or a switch.
  - Multiplicate Decrease:: TCP reacts to a timeout by halving cwnd.
- Although cwnd is defined in bytes, the literature often discusses congestion control in terms of packets (or more formally in Maximum Segment Size(MSS)).
- cwnd is not allowed below the size of a single packet

AIMD (Additive Increase / Multiplicative Decrease)

- It has been shown that AIMD is a necessary condition for TCP congestion control to be stable.
- Because the simple CC mechanism involves timeouts that cause retransmissions, it is important that hosts have an accurate timeout mechanism.
- Timeouts set as a function of average RTT and standard deviation of RTT.
- However, TCP hosts only sample round-trip time once per RTT using coarse-grained clock.

AIMD (Additive Increase / Multiplicative Decrease)



Slow Start

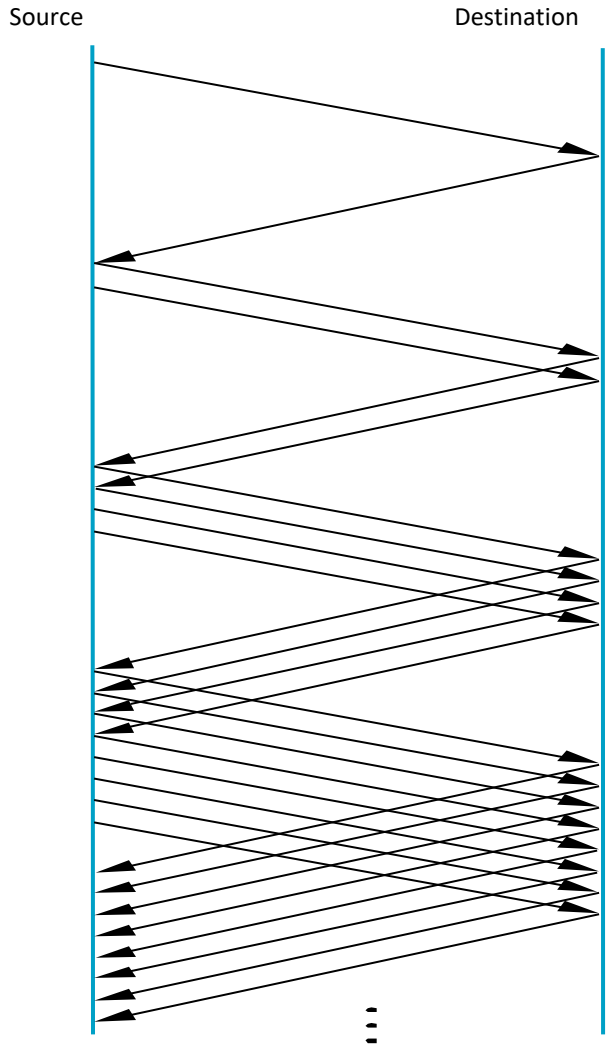
- Linear additive increase takes too long to ramp up a new TCP connection from cold start.
- Beginning with TCP Tahoe, the slow start mechanism was added to **provide an initial exponential increase in the size of cwnd**.
- Remember mechanism by:
  - *slow start prevents a slow start*. Moreover, slow start is slower than sending a full advertised window's worth of packets all at once.

Slow Start

- The source starts with cwnd = 1.
- Every time an ACK arrives, cwnd is incremented.
  ➔cwnd is effectively doubled per RTT time.

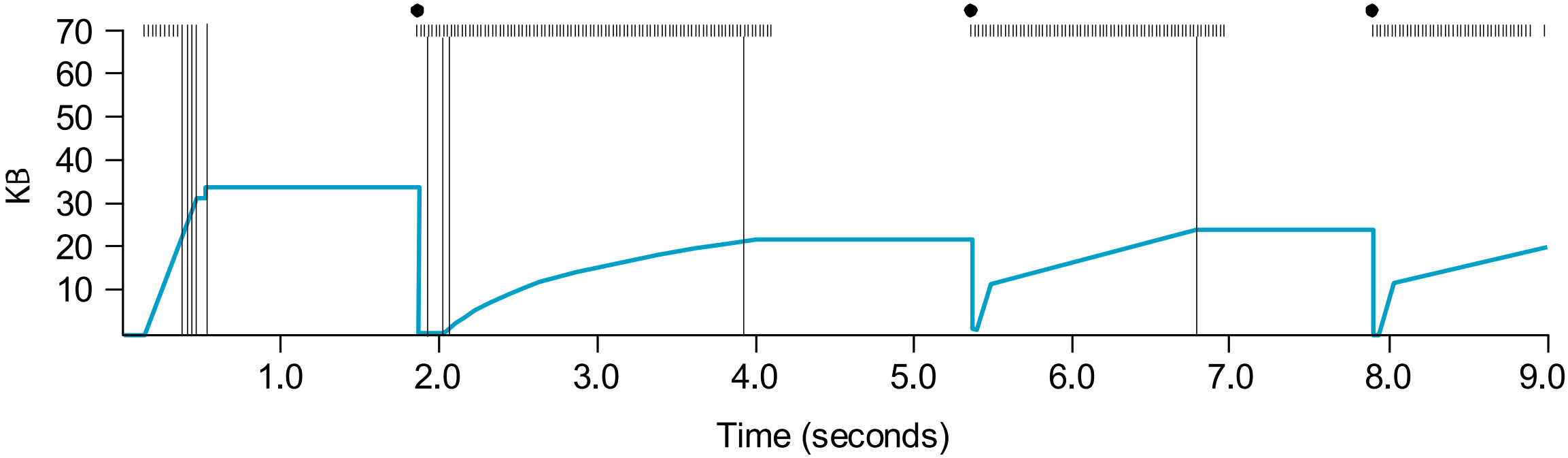
- Two slow start situations:
  - At the very beginning of a connection {cold start}.
  - When the connection goes dead waiting for a timeout to occur (i.e, the advertized window goes to zero!)

- Strategy: Slow Start
  Add one packet per ACK



Slow Start

- However, in the second case the source has more information. The current value of cwnd can be saved as a **congestion threshold**.
- This is also known as the “slow start threshold” **ssthresh**.


70
60
50
40
30
20
10
KB
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
Time (seconds)

Fast Retransmit

- Coarse timeouts remained a problem, and Fast retransmit was added with TCP Tahoe.
- Since **the receiver responds every time a packet arrives**, this implies the sender will see duplicate ACKs.

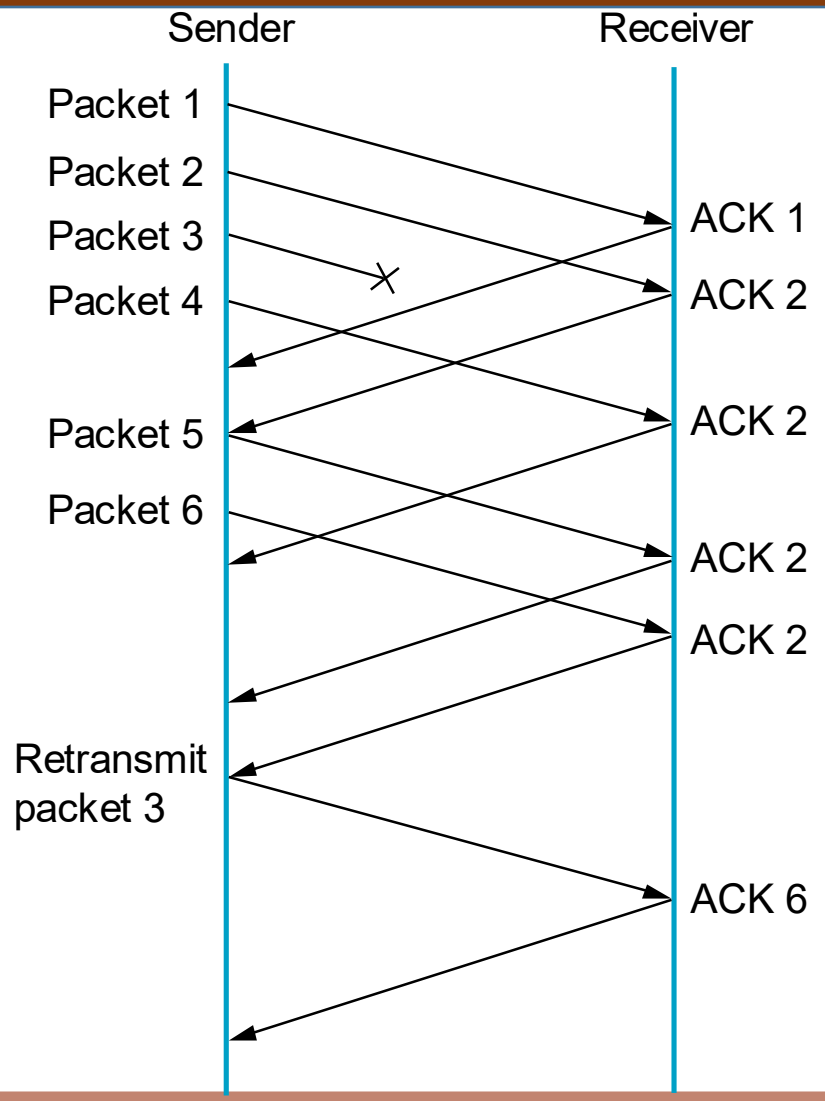
Basic Idea:: *use **duplicate ACKs** to signal lost packet*

- **Fast Retransmit**
- **Upon receipt of *three* duplicate ACKs, the TCP Sender retransmits the lost packet.**

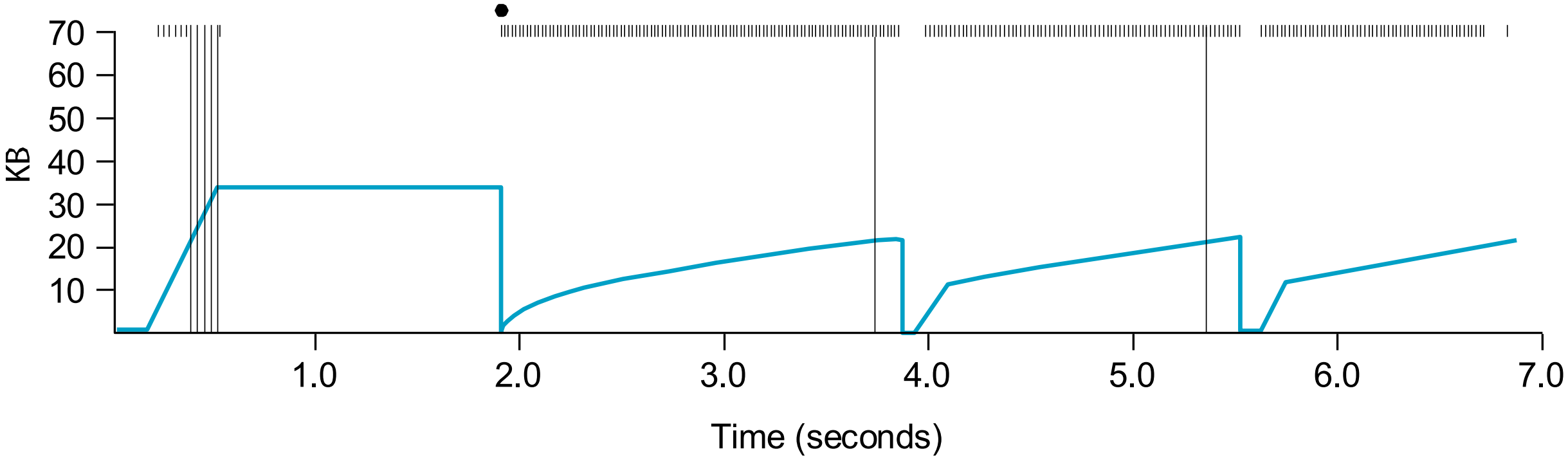
Fast Retransmit

- Generally, fast retransmit eliminates about half the coarse-grain timeouts.
- This yields roughly a **20% improvement in throughput.**
- Note – fast retransmit does not eliminate all the timeouts due to small window sizes at the source.

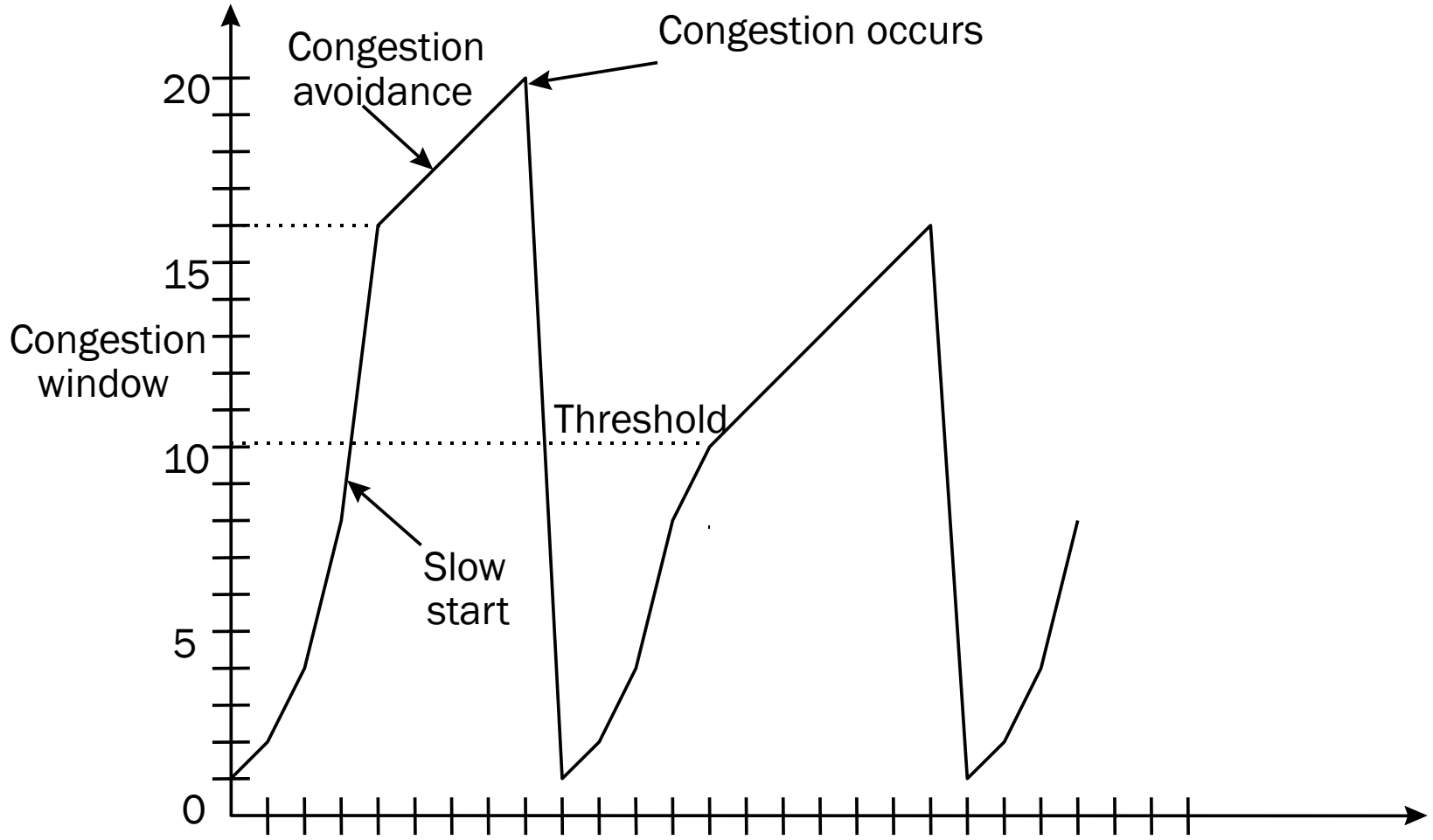
- Fast Retransmit based on three duplicate ACKs



TCP Fast Retransmit Trace



TCP Congestion Control



Fast Recovery

- Fast recovery was added with TCP Reno.

- Basic idea:: When fast retransmit detects three duplicate ACKs, start the recovery process from congestion avoidance region and use ACKs in the pipe to pace the sending of packets.

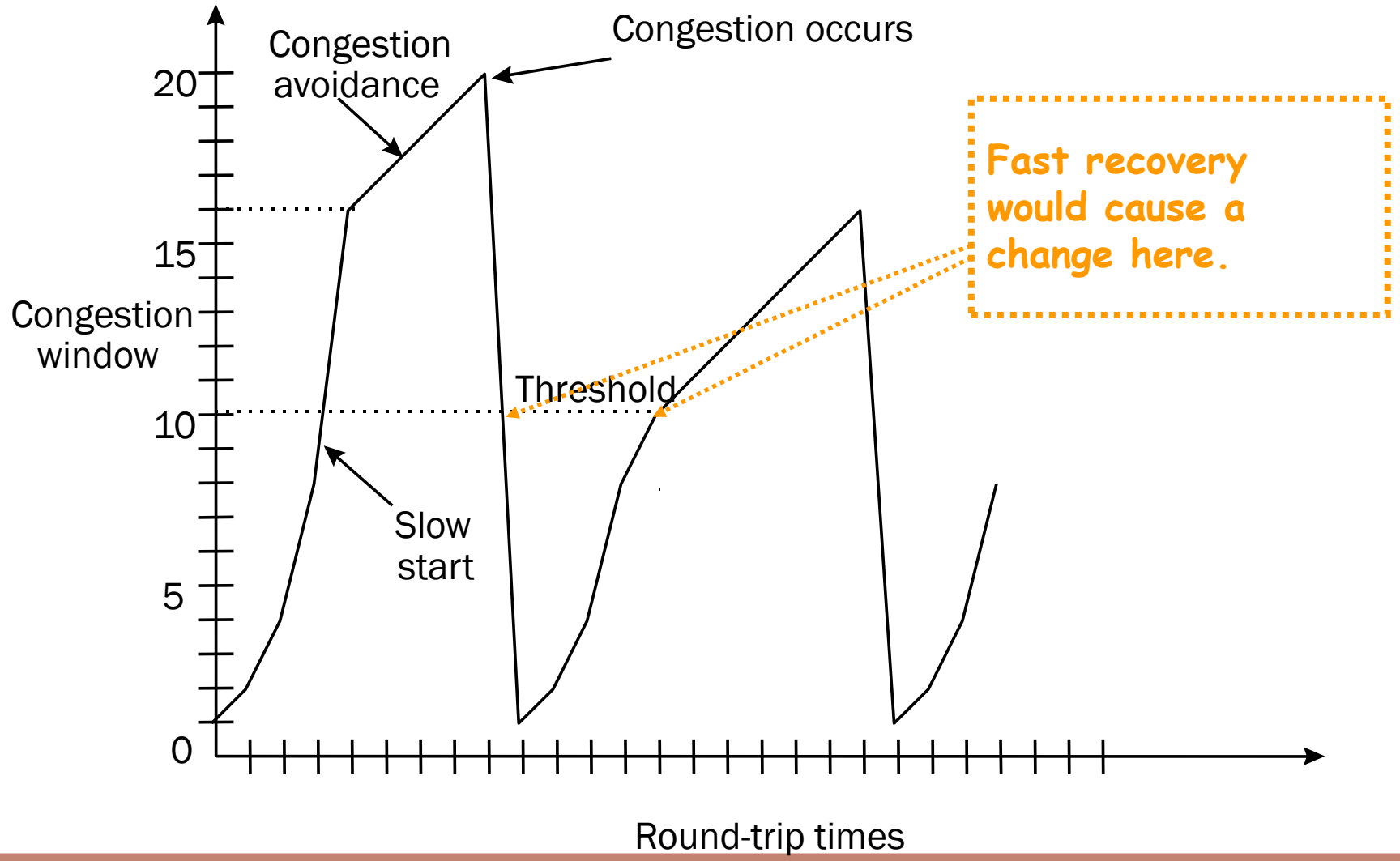
- Fast Recovery
  - After Fast Retransmit, **half cwnd** and commence recovery from this point using **linear additive increase** 'primed' by left over ACKs in pipe.

Modified Slow Start

- With fast recovery, slow start only occurs:
  - At cold start
  - After a coarse-grain timeout

- *This is the difference between TCP Tahoe and TCP Reno!!*

TCP Congestion Control



Bibliography

- Apontamentos de redes de Computadores, André Moreira
- https://www.imperva.com/learn/application-security/osi-model/
- https://developer.ibm.com/articles/au-tcpsystemcalls/
- https://www.geeksforgeeks.org/udp-server-client-implementation-c/
- Lecture material taken from “Computer Networks A Systems Approach”, Third Ed.,Peterson and Davie, Morgan Kaufmann, 2003

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Rua Dr. António Bernardino de Almeida, 431

4249-015 Porto, Portugal

—

T(+351) 228 340 500

F (+351) 228 321 159

—

www.isep.ipp.pt

www: moodle.isep.ipp.pt

http://www.isep.ipp.pt/mescc

e-mail: llf@isep.ipp.pt