Computer Networks
Chapter 5
End-to-End protocols

Prof. Marino Miculan

Chapter Outline and Goals

- Simple Demultiplexer (UDP)
- Reliable Byte Stream (TCP)

- Chapter Goals
  - Understanding the demultiplexing service
  - Discussing simple byte stream protocol

Introduction to End-to-end Protocols

Limitations of Network Protocols

↳ dato che il liv. 3 è best-effort, TCP si occupa di rendere il flusso dei messaggi affidabile

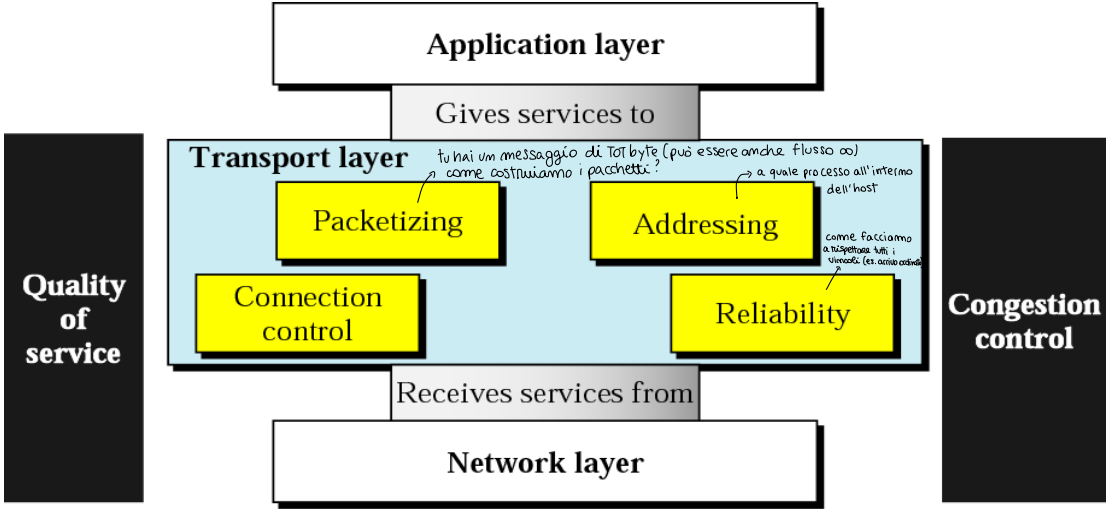
- Network protocols implement a host-to-host, best-effort service
- Typical limitations of the network on which transport protocol will operate
  - Drop messages
  - Reorder messages
  - Deliver duplicate copies of a given message
  - Limit messages to some finite size
  - Deliver messages after an arbitrarily long delay
- Usually, end-to-end (i.e., application) programmers are not satisfied with this

End-to-end Protocols

- Common properties that a transport protocol can be expected to provide
  - Guarantees message delivery
  - Delivers messages in the same order they were sent
  - Delivers at most one copy of each message
  - Supports arbitrarily large messages
  - Supports synchronization between the sender and the receiver
  - Allows the receiver to apply flow control to the sender
  - Supports multiple application processes on each host
- Challenge for Transport Protocols
  - **Develop algorithms that turn the less-than-desirable properties of the underlying network into the high level of service required by application programs**

↳ simile a Sliding window
↳ es. Ethernet e IP non ce hanno già integrato
↳ es. negli EMBEDDED (Arduino) gira un solo programma
nei nostri PC ci sono tanti processi in parallelo, quando arriva un pacchetto IP, a quale processo è destinato? come identifico il processo? (PID su Linux)

Position of the transport layer

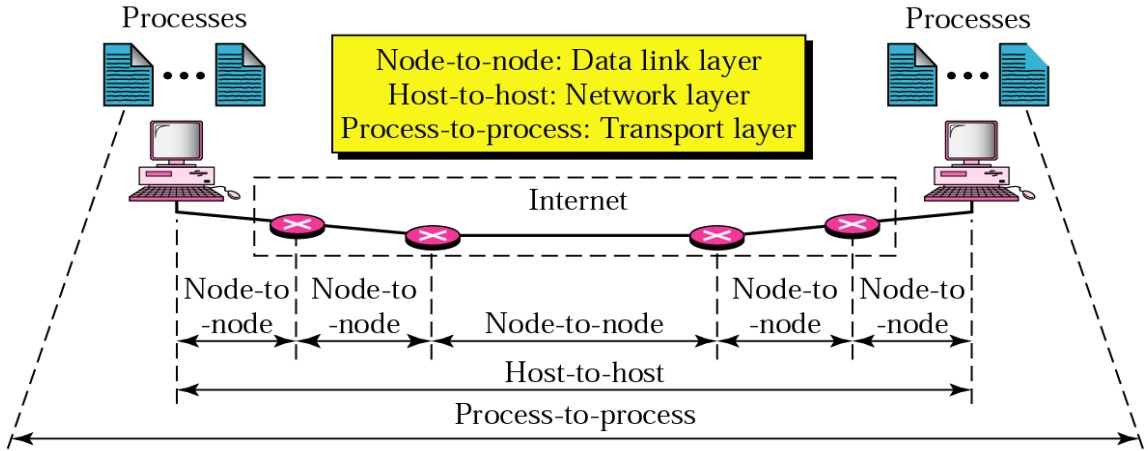


- Some functionalities (FUNZIONALITÀ CROSS LAYER) (such as QoS and Congestion Control) do not fit a single layer, but rather require the collaboration between layers.

Position of the transport layer

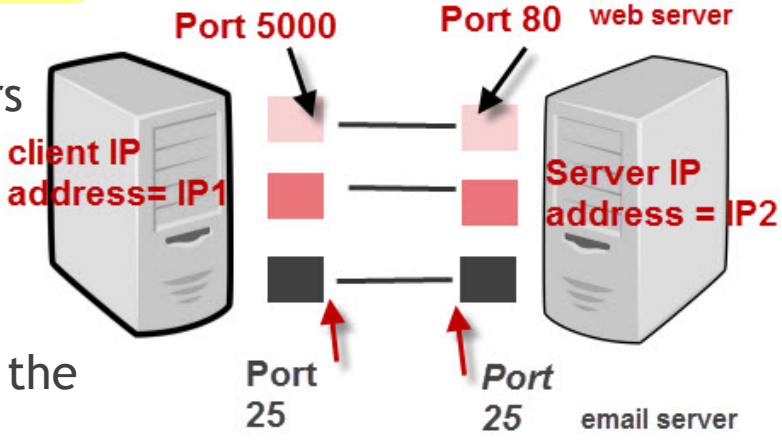
LIV 2 : indirizzi INTERFACCIA - INTERFACCIA
LIV 3 : indirizzi HOST - HOST
LIV 4 : indirizzi PROCESSO nell'HOST - PROCESSO nell'HOST

- End-to-end protocols implement process-to-process communications
  - Can be *connection-oriented* or *connectionless*
  - Many endpoints inside a node => **Another level of addressing is needed, in order to identify them**



End-to-end addressing in TCP/IP

- Internet transport layer endpoints are **ports** → si potrebbero usare i PID se tutti i S.O. standardizzassero i numeri dei PID
- A port number is a number between 0 and 65535 (16 bits) → per ogni IP ci sono 65K porte usabili
  1 IP N PORTE
- In each connections, two port numbers are involved, one for each process
  - a port number on the host of process A, a port number on the host of process B
  - Not needed to be the same numbers!
- plus, of course, the host addresses of the hosts the processes are running on
  - Many processes on one host => many ports on the same network address

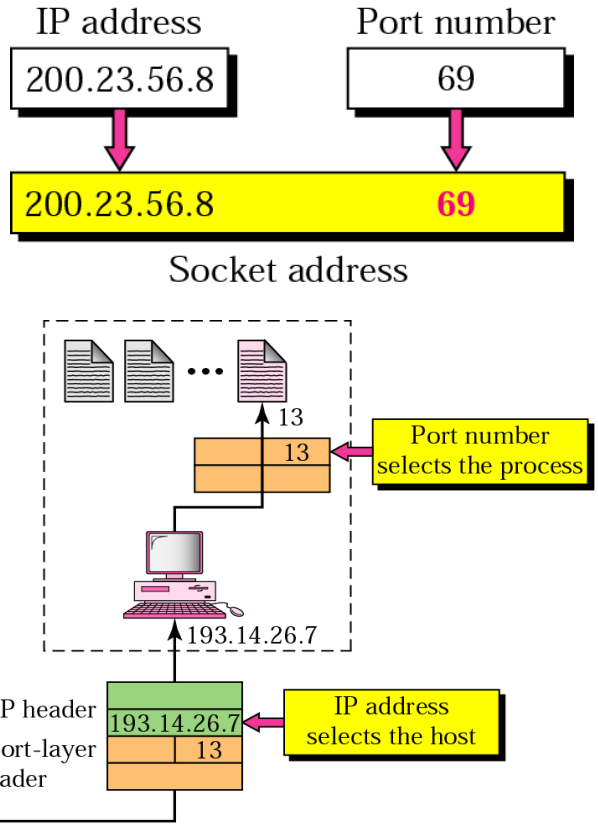


Socket

con NAT/PAT traducevamo le porte, quindi il router ha a disposizione 65K porte, quindi nella LAN posso avere 65K processi => problema del NAT

- A communication endpoint is called **socket**
  - Created by means of suitable system calls
- its address is the pair <IP, port>
  - The IP address identifies the host, the port number identifies the process within that host
- Two processes, to communicate, must have a socket each, and use the same transport protocol
- At each instant, **a communication is completely identified by the tuple**

<IP_A, $port_A$, IP_B, $port_B$, protocol>

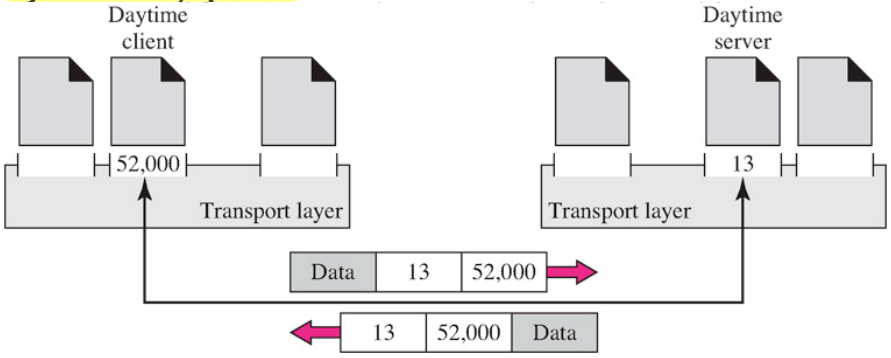


Client/server model

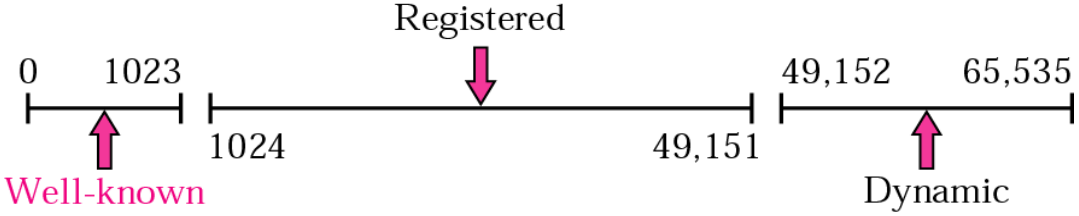
- Often two process communicate according to the **client/server** model
- **Server**
  - A process which offers a service (e.g. access to a resource)
  - Always in execution (when service is offered)
  - Waiting for client's requests (*passive* role)
- **Client**
  - Willing to use the service
  - May be active only when needed
  - Places requests to the server (*active* role)
- Notice: clients and servers are **processes**, not hosts

Ports for client-server model

- To start the communication, client must be able to know the port of the server
- Server can learn client's address when she is contacted
- Often, server uses **well-known port** (↳ definite da uno STANDARD)
  - Es: server Web: port 80 (HTTP); server email: 25
- Client uses **ephemeral** (PORTE EFFIMERE), or dynamic, ports
  - Randomly chosen, and possibly only for each communication



IANA ranges

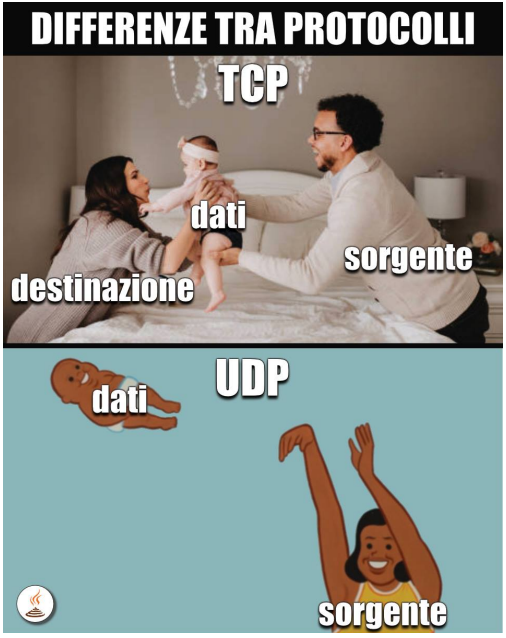


- **Well-known ports**: used by official servers. Can be used by only admin-level process (guarantee a limited form of server authentication). → possono essere usate soltanto da processi con permessi di amministratore → se mi collego ad un srv con porta < 1024 allora so che è stato avviato da un amministratore
- **Registered ports**: may have a standard usage, but not restricted to admin processes
- **Dynamic**: free for all; usually allocated by the kernel to client which do not specify any port
- See /etc/services and RFC 1700

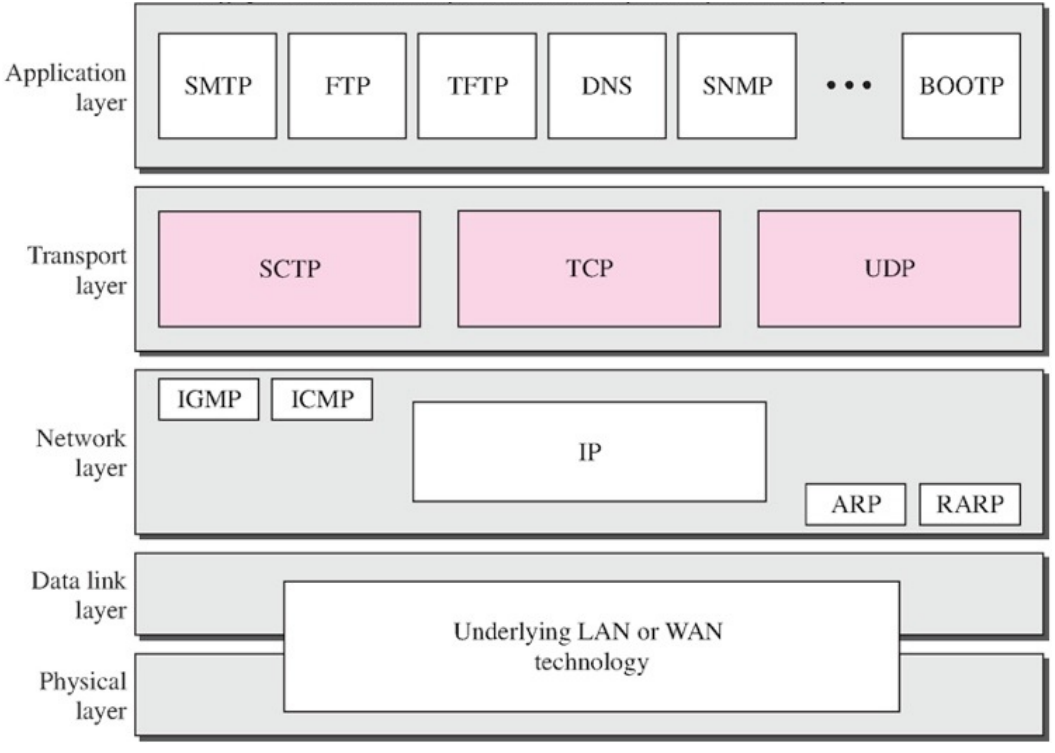
↓
se non ti fidi allora la macchina non è affidabile

Two kinds of communications

- **Connection oriented** → creo un servizio stabile basandomi su uno instabile (IP)
  - A connection must be established before exchanging data => more expensive (in termini di banda, memoria)
  - Datagrams belong to one connection, hence more under control => more reliable
  - In TCP/IP: implemented by **TCP** (and SCTP)
- **Connectionless** → replichiamo il liv. 3, quello che arriva dalla rete lo inoltriamo alle applicazioni, con l'aggiunta delle porte (basso overhead)
  - Each message (*datagram*) is sent without an explicit connection opening => faster response
  - Each datagram is independent
  - Usually unreliable (same kind of service of IP)
  - In TCP/IP: implemented by **UDP**
    - es. Teams, che usa UDP, dato che i pacchetti possono arrivare disordinati, questi non vengono ordinati da UDP (a differenza di TCP che lo fa), ci pensa l'applicazione



Position of transport protocols



Other kinds of communications

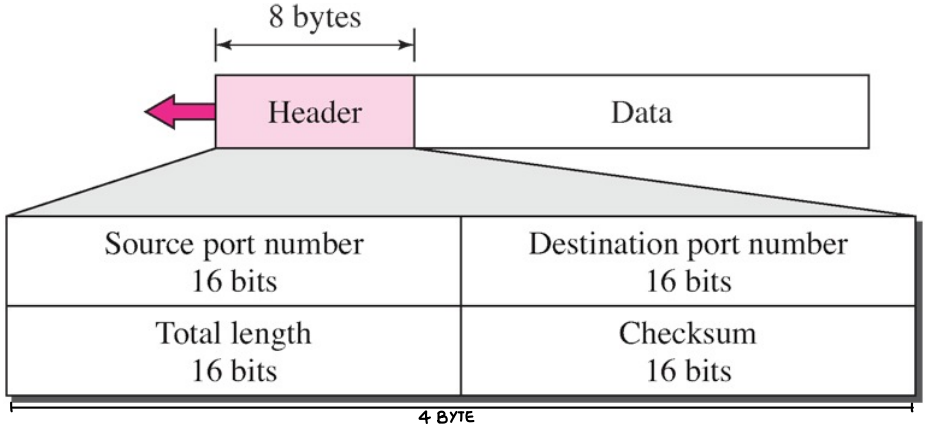
- **Reliable connectionless**
  - Theoretically possible, rarely implemented
  - Can be simulated using TCP, or adding controls to UDP
  - In TCP/IP: RUDP, RDP (not widespread)
- **Request/Reply**
  - Each request from the client is followed by a corresponding reply from the server
  - Each request/reply is independent from the others
  - In TCP/IP: implemented in application level protocols, e.g. RPC, RMI, SOAP, REST...

User Datagram Protocol (UDP)

User Datagram Protocol (UDP): Simple Demultiplexer

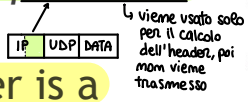
si appoggia ad IP, utilizza IP aggiungendo poco OVERHEAD

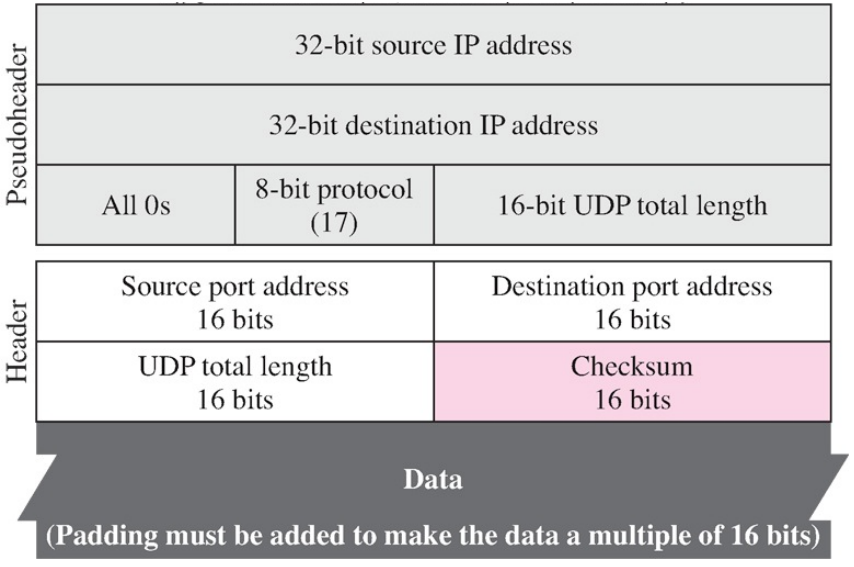
- Extends host-to-host delivery service of the underlying network into a process-to-process communication service
- Adds a level of demultiplexing which allows multiple application processes on each host to share the network
- No guarantees beyond those of IP (apart some buffering)
- UDP datagram format:
- Total length could be inferred from network header, but better have it also here



Checksum calculation

- Checksum, if present (i.e. non zero), is computed over UDP header + data + *pseudo-IP header*
- The pseudo-IP header is a "summary" of the real IP header, containing only:
  - IP addresses
  - protocol (17 for UDP)
  - Total IP datagram length
  - Other data (fragmentation, TTL, etc) are omitted

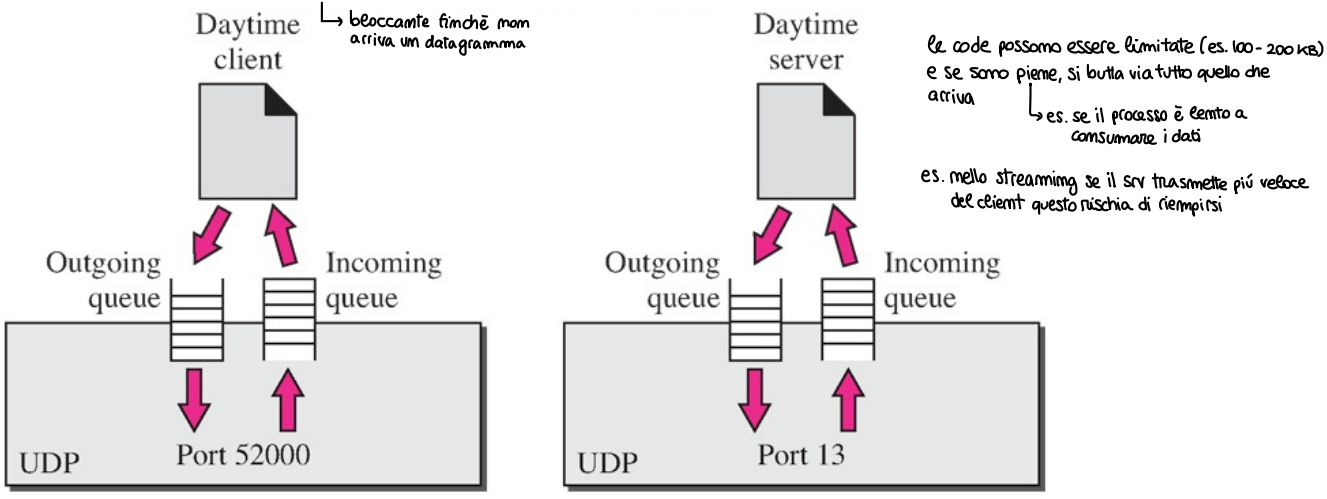




Se durante la comunicaz. viene cambiato IP (es. NAT), devo scegliere cosa fare con il checksum: - lo ricalcolo (spreco tempo) - non lo uso più (lo setto a 0)

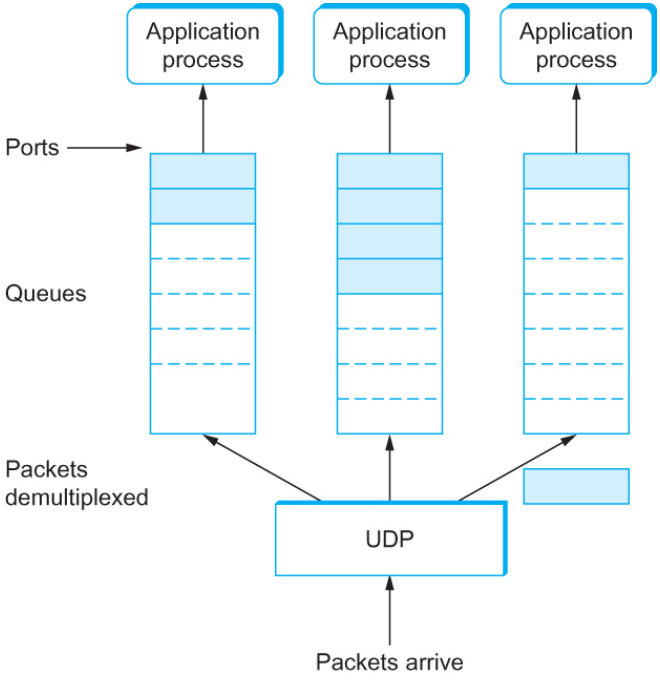
API for UDP services

- When bound to a port, each socket has an input and output queue, maintained in kernel → BUFFER, CODE di DATAGRAMMI
- Queues are accessed by the applications by calling syscalls like `sendmsg()`, `recvmsg()`



Simple Demultiplexer (UDP)

- As they arrive to the host, datagrams are checked and enqueued on the queue associated to the given port
- Application processes consume packets in FIFO order, and at a whole packet at a time
- If a queue has no free space, the datagram is silently discarded
- If the port is not associated to a queue (i.e. no process is using the port), the datagram is discarded
- Datagrams from different sources to the same port can be interleaved

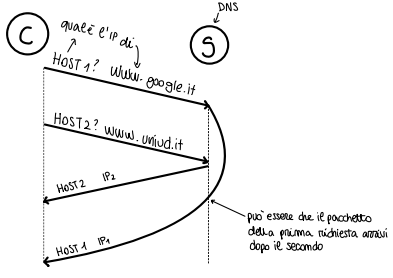


Features of UDP

- Connectionless communication
- Datagram are not numbered, do not belong to a session/ connection
  - independently delivered
- No error control, no flow control
  - sender is not notified if datagrams are lost, or the receiver is congested
- Just encapsulation in IP
  - possibly with fragmentation

UDP is...

- **transaction-oriented**, suitable for simple query-response protocols such as the Domain Name System or the Network Time Protocol.
- **simple**, suitable for bootstrapping or other purposes without a full protocol stack, such as the DHCP and TFTP.
- **stateless**, suitable for very large numbers of clients, such as in streaming media applications for example IPTV



nella risposta mando anche

UDP is suitable for...

- **Real-time applications** such as Voice over IP, online games (where it may be better to lose one datagram, than delaying the delivery of data to processes for waiting missing ones)
- **Unidirectional communication**, e.g. for broadcast and multicast communications, like service discovery and shared information
- Implementing specific end-to-end transport protocols in user-space (like Google's QUIC, Sun's RPC)

Some UDP-based protocols

Port	Protocol	Description
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
53	Nameserver	Domain Name Service
67	Bootps	Server port to download bootstrap information
68	Bootpc	Client port to download bootstrap information
69	TFTP	Trivial File Transfer Protocol
111	RPC	Remote Procedure Call
123	NTP	Network Time Protocol
161	SNMP	Simple Network Management Protocol
162	SNMP	Simple Network Management Protocol (trap)

Transport Control Protocol (TCP)

A Protocol for Packet Network Intercommunication

VINTON G. CERF AND ROBERT E. KAHN,
MEMBER, IEEE

Abstract — A protocol that supports the sharing of resources that exist in different packet switching networks is presented. The protocol provides for variation in individual network packet sizes, transmission failures, sequencing, flow control, end-to-end error checking, and the creation and destruction of logical process-to-process connections. Some implementation issues are considered, and problems such as internetwork routing, accounting, and timeouts are exposed.

INTRODUCTION

IN THE LAST few years considerable effort has been expended on the design and implementation of packet switching networks [1]-[7],[14],[17]. A principle reason for developing such networks has been to facilitate the sharing of computer resources. A packet communication network includes a transportation mechanism for delivering data between computers or between computers and terminals. To make the data meaningful, computer and terminals share a common protocol (i.e, a set of agreed upon conventions). Several protocols have already been developed for this purpose [8]-[12],[16]. However, these protocols have addressed only the problem of communication on the same network. In this paper we present a protocol design and philosophy that supports the sharing of resources that exist in different packet switching networks.

of one or more *packet switches*, and a collection of communication media that interconnect the packet switches. Within each HOST, we assume that there exist *processes* which must communicate with processes in their own or other HOSTS. Any current definition of a process will be adequate for our purposes [13]. These processes are generally the ultimate source and destination of data in the network. Typically, within an individual network, there exists a protocol for communication between any source and destination process. Only the source and destination processes require knowledge of this convention for communication to take place. Processes in two distinct networks would ordinarily use different protocols for this purpose. The ensemble of packet switches and communication media is called the *packet switching subnet*. Fig. 1 illustrates these ideas.

In a typical packet switching subnet, data of a fixed maximum size are accepted from a source HOST, together with a formatted destination address which is used to route the data in a store and forward fashion. The transmit time for this data is usually dependent upon internal network parameters

IEEE Trans on Comm., 1974 https://bit.ly/2QbDPpi

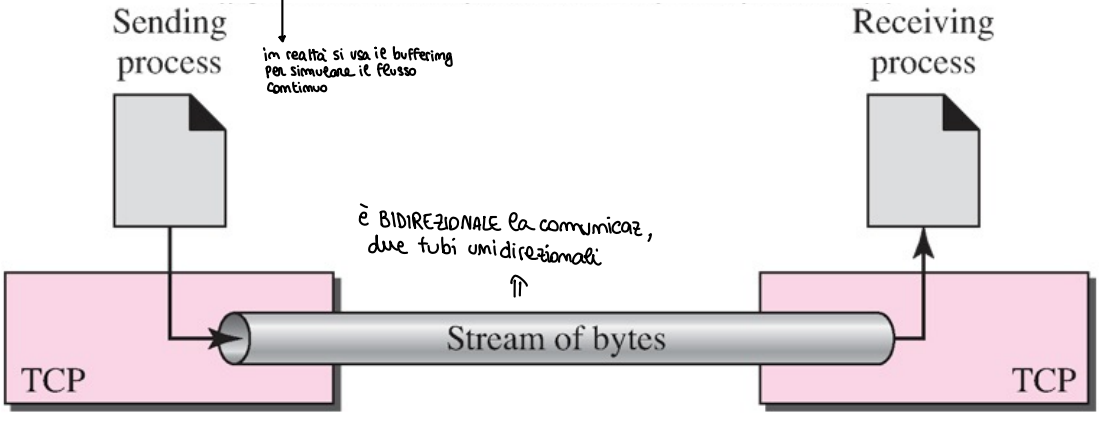
Transport Control Protocol (TCP): Reliable Byte Stream

- In contrast to UDP, Transmission Control Protocol (TCP) offers the following services
  - Reliable
  - Connection oriented
  - Byte-stream service

stabilisce una PIPE (flusso) tra due socket
↳ il SENDER scrive una cosa sul tubo e l'altro legge → arriva tutto e senza duplicati
Si basa su IP, quindi dei pezzi possono non arrivare o arrivare in disordine → li risolve TCP

non avviene la connessione vera e propria, si mandano pacchetti IP e si usano tecniche di simcronizzazione

flusso continuo, non c'è una struttura o una dimensione



Flow control VS Congestion control

più difficile da gestire

FLUSSO

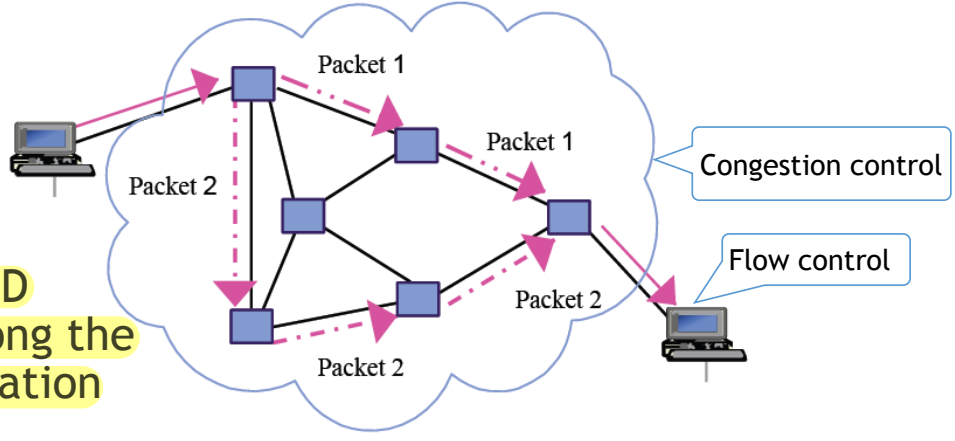
chi riceve non riesca a stare al passo di chi manda

- **Flow control** involves preventing sender from overrunning the capacity of the *receiver*

troppi dati e rischio di saturare la rete che deve attraversare (su cui non ho controllo, non so chi sono i router intermedi)

- **Congestion control** involves preventing too much data from being injected into the *network*, thereby causing switches or links to become overloaded
- TCP has to deal with both problems
- As a result, end-to-end bandwidth and delays are defined by the receiver AND the slowest link/switch along the path from source to destination

COLLO DI BOTTIGLIA



End-to-end Issues

a differenza di Sliding Window di liv. 2 dove c'è una commessione PUNTO-PUNTO e entrambi gli host hanno lo stesso RTT, Nel TCP ci sono RTT che possono essere molto diversi e la commessione avviene tramite Internet

- At the heart of TCP is the **sliding window algorithm** (see Chapter 2)
- Issues to be addressed by the sliding window algorithm, as TCP runs over the Internet rather than a point-to-point link:
  - TCP supports **logical connections** between processes that are running on two different computers in the Internet
    - in slidingwindow c'era già una commessione, qui con TCP va prima stabilita
  - TCP connections are likely to have **widely different RTT** times (ranging from 10 μs to 10 s – six orders)
    - in SLI.WIN c'era lo stesso RTT, qui no. Va gestito bene il timeout
    - TEMPO VARIABILE, percorsi non fissi
  - Packets may get **reordered** in the Internet (usually this does not happen on a single link)
    - nello SLIDING WINDOWS i pacchetti arrivavano ordinati, con TCP (basandosi su IP) l'ordine non è garantito

End-to-end Issues

- For **flow control**: TCP needs a mechanism using which each side of a connection will learn what resources **the other side** is able to apply to the connection
- For **congestion control**: TCP needs a mechanism using which the sending side will learn the capacity of the **network**
- We will see flow control first; congestion control in the next chapter
- But before going into details, we need to know the basic data and protocol

TCP Segment

- TCP is a **byte-oriented** protocol, which means that the sender writes bytes into a TCP connection and the receiver reads bytes out of the TCP connection
- No message boundaries are kept
  - Other protocols do, e.g. SCTP
- Although "byte stream" describes the service TCP offers to application processes, TCP does not, itself, transmit individual bytes over the Internet

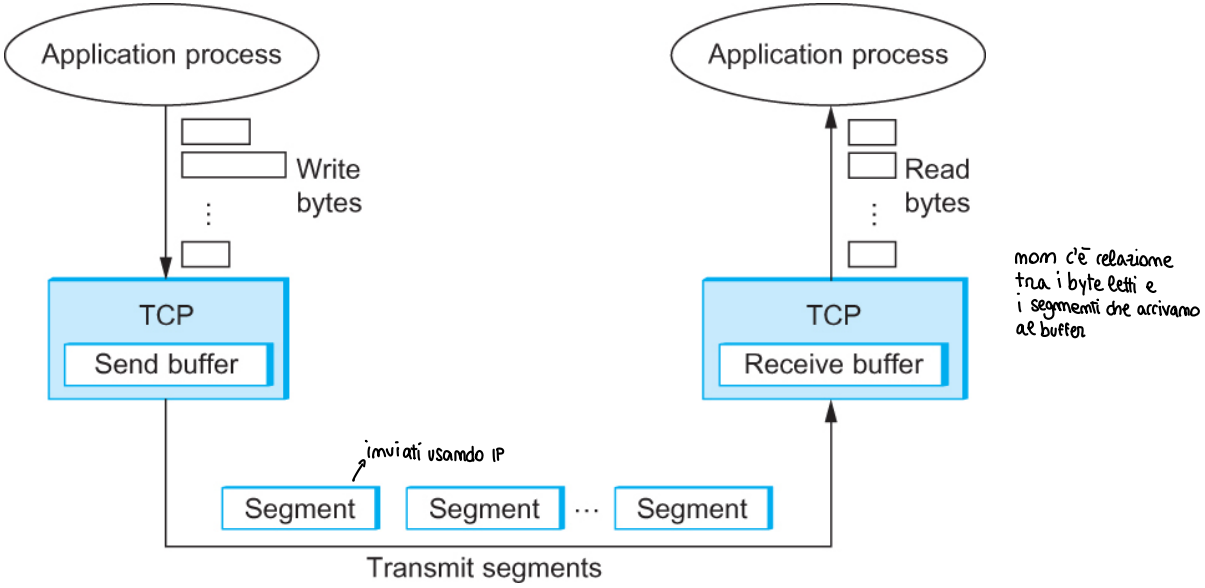
TCP Segment

TCP non invia singoli byte, memorizza un numero sufficiente di byte e poi invia

- on the source host, TCP 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.
- on the destination host, TCP empties the contents of the packet into a receive buffer, and the receiving process reads from this buffer at its leisure.
- The packets exchanged between TCP peers are called **segments**.

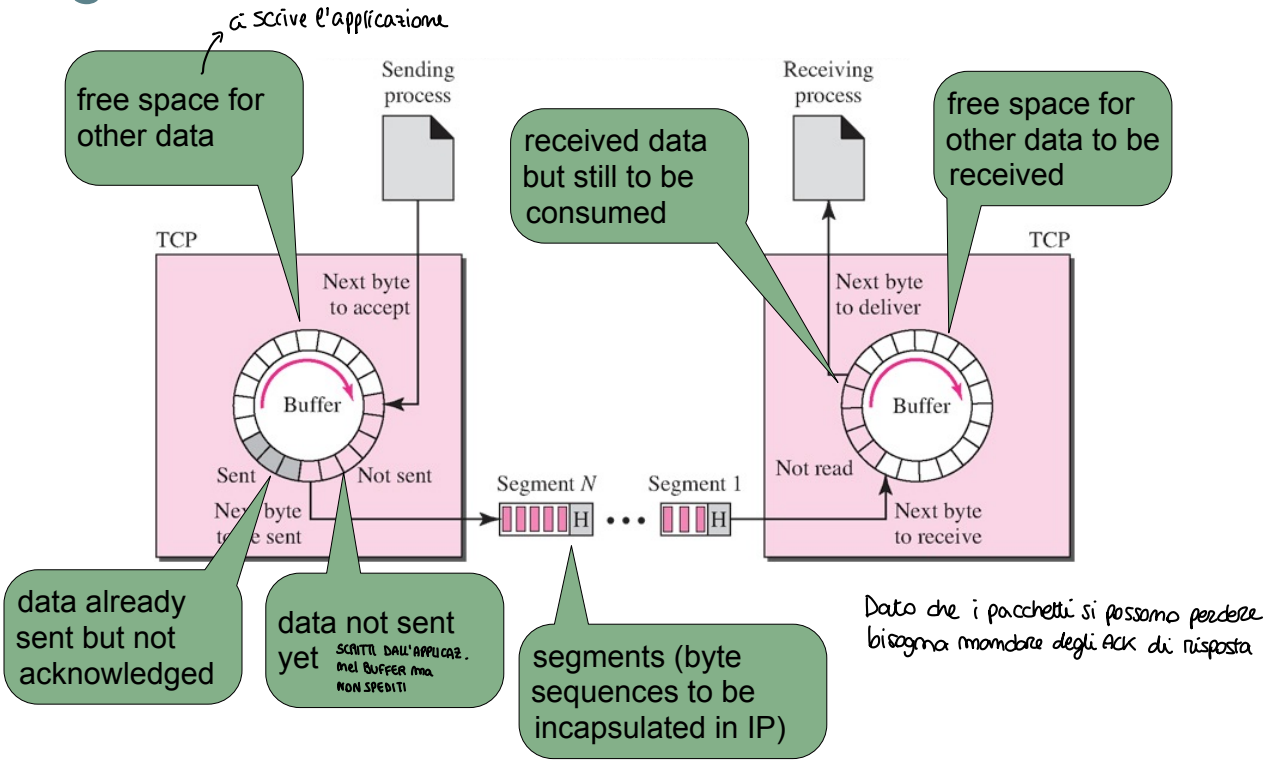
TCP Segment

How TCP manages a byte stream



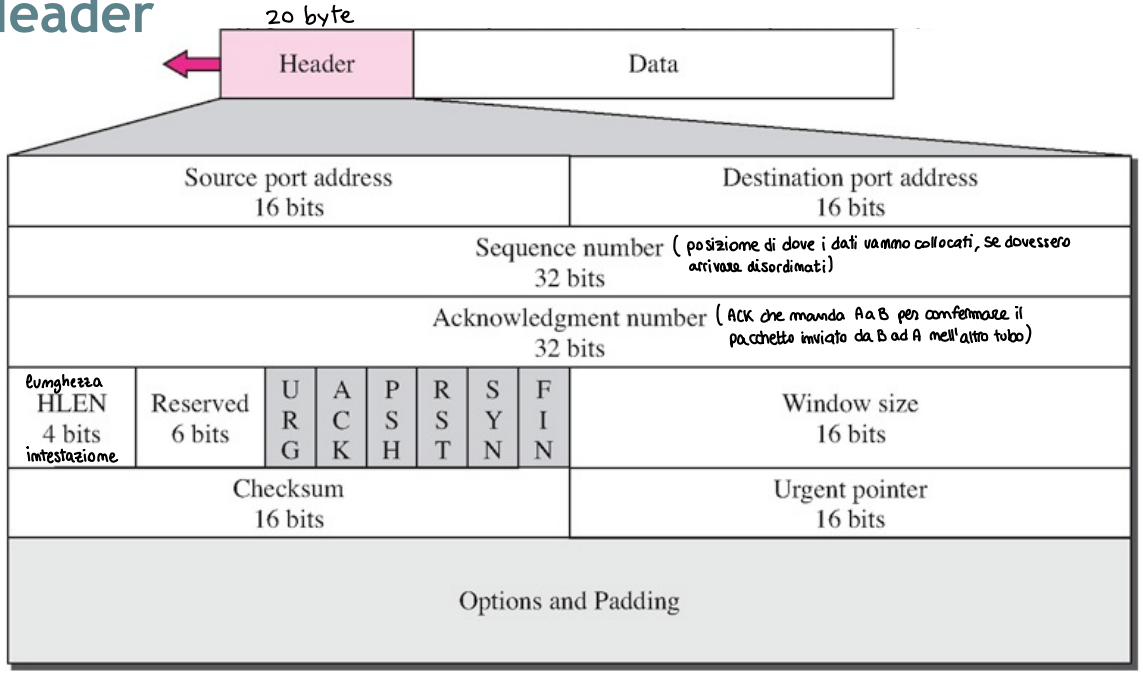
TCP Segment

ci sono due tubi opposti, ovvero TX → RX, RX ← TX che sono indipendenti e ci sono 4 buffer (due per ogni processo)



Dato che i pacchetti si possono perdere bisogna mandare degli ACK di risposta

TCP Header



TCP Header Format

TCP Header

- The SrcPort and DstPort fields identify the source and destination ports, respectively.
- The Acknowledgment, SequenceNum, and AdvertisedWindow fields are all involved in TCP's sliding window algorithm (we will see later).
- Because TCP is a byte-oriented protocol, *each byte of data has a sequence number*; the SequenceNum field contains the sequence number for the first byte of data carried in that segment.
- The Acknowledgment and AdvertisedWindow fields carry information about the flow of data going *in the other direction (piggybacking)*.

TCP Header

- The 6-bit Flags field is used to relay control information between TCP peers.
- The possible flags include SYN, FIN, RESET, PUSH, URG, and ACK.
- The SYN and FIN flags are used when establishing and terminating a TCP connection, respectively.
- The ACK flag is set any time the Acknowledgment field is valid, implying that the receiver should pay attention to it.

TCP Header

i pacchetti IP sotto non hanno la priorità, sono cose che si vedono solo a liv. 4, i pacchetti IP sono tutti uguali

- The URG flag signifies that this segment contains urgent data. When this flag is set, the UrgPtr field indicates where the nonurgent data contained in this segment begins.
  - The urgent data is contained at the front of the segment body, up to and including a value of UrgPtr bytes into the segment.
- The PUSH flag signifies that the sender invoked the push operation, which indicates to the receiving side of TCP that it should notify the receiving process of this fact.

TCP Header

- The RESET flag signifies that the receiver has become confused, it received a segment it did not expect to receive—and so wants to abort the connection.
- Finally, the Checksum field is used in exactly the same way as for UDP—it is computed over the TCP header, the TCP data, and the pseudoheader, which is made up of the source address, destination address, and length fields from the IP header.

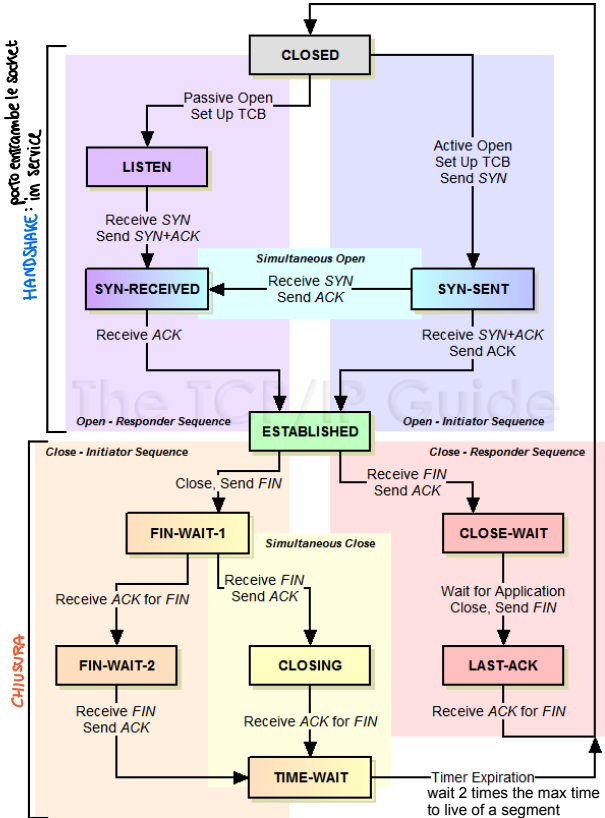
State diagram of TCP

- A connection-oriented communication must keep some information about the data, the missing segments, etc.
- TCP is **stateful** → tipo um AUTOMA
- Each TCP socket is in a state
- Transitions are caused by **events**:
  - Execution of application-level commands on the socket
  - Reception of segments with suitable flags
- Transitions can cause segments with suitable flags to be sent

State diagram of TCP

→ogni socket ha il suo automa

- States and state transitions of a TCP socket are defined by the protocol by means of a DFA (actually a Mealy machine)
- Each transition is triggered by some event, and may yield the transmission of a segment (with suitable flags)
- Three phases:
  1. Handshaking: creation of connection
  2. Data transfer: when data flows
  3. Closing: end of connection
- A connection can stay in ESTABLISHED for unlimited amount of time (even forever)

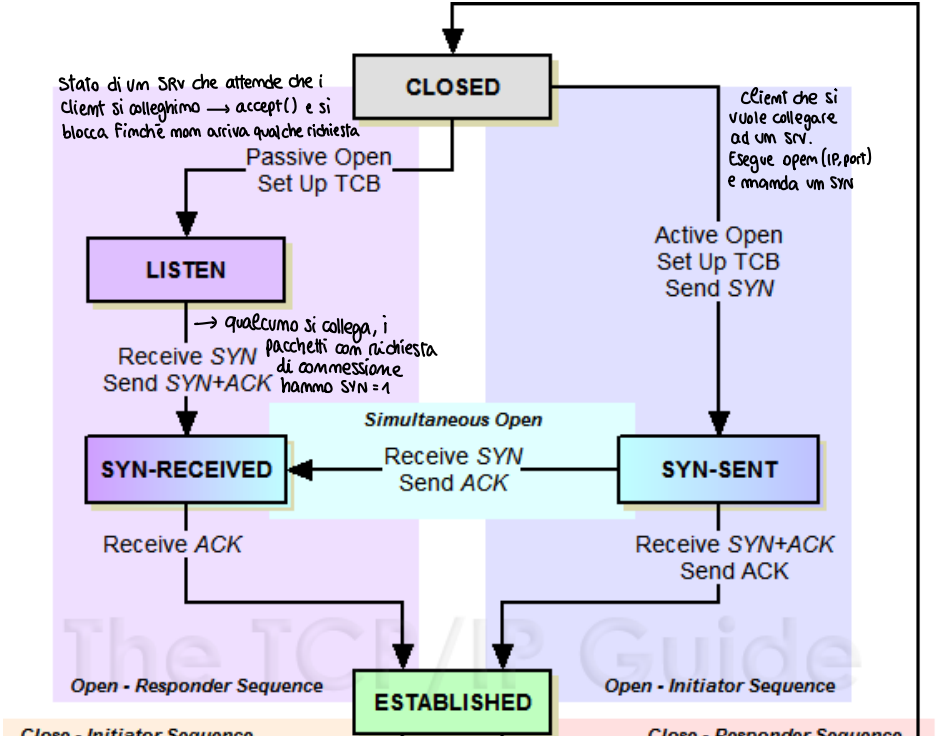


State diagram of TCP - Phase 1: handshake

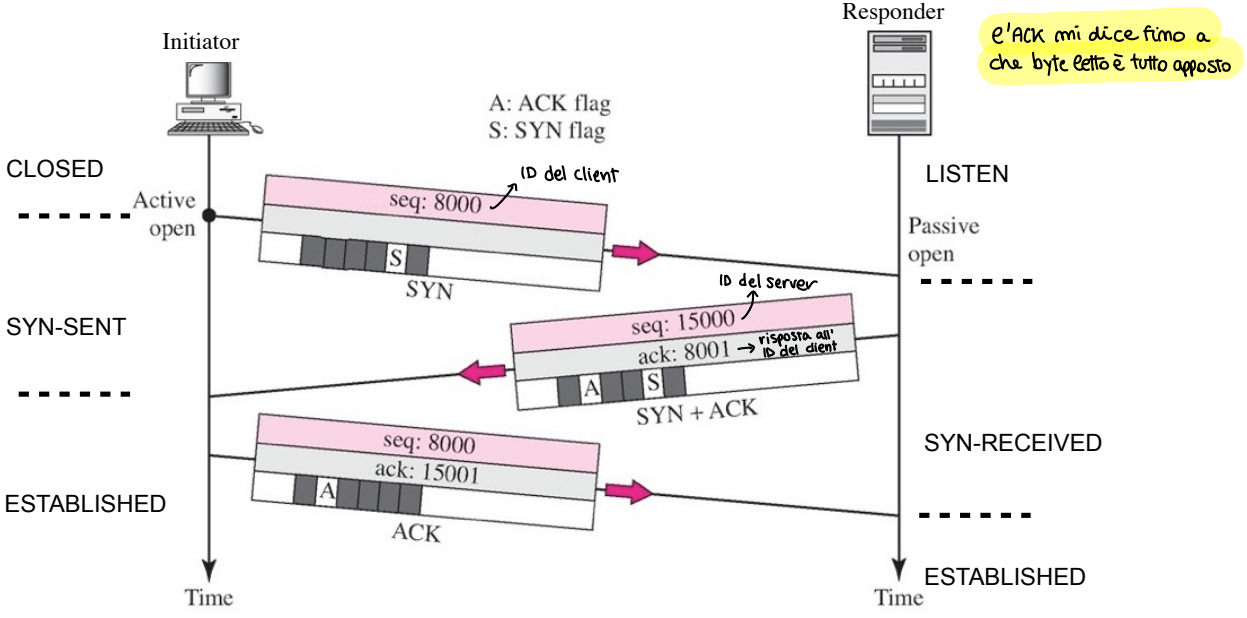
- Before sending any data, two processes must establish a **connection** (condividere uno stato) ⟶ un processo fa la accept() e l'altro fa la connect()
  (accept() è BLOCCANTE)
  - A process (the *responder*, or *server*) waits for a connection, by executing a suitable TCP function for the *passive opening*
    - On *nix: accept(). Blocking syscall.
  - The other process (the *initiator*, or *client*) begins the connection by executing the *active opening*
    - On *nix: connect(). Time-outed syscall.
  - At this point, the two TCP layers execute the *three-way handshake*

CLIENT: entità ATTIVA
SERVER: entità PASSIVA

State diagram of TCP - Phase 1: handshake



State diagram of TCP - Phase 1: handshake



Timeline for three-way handshake algorithm

State diagram of TCP - Phase 1: handshake

1. host in passive open (server) is waiting
2. when client executes active open, sends first segment (SYN) which has starting sequence number (and no payload)
3. server replies with SYN+ACK: acknowledge client's sequence number and sets her sequence number (again no payload)
4. client answers with ACK: acknowledge server's sequence number (still no payload)
5. now the connection is established and segments carrying payloads of real data can be sent in both directions

State diagram of TCP - Phase 1: particular cases

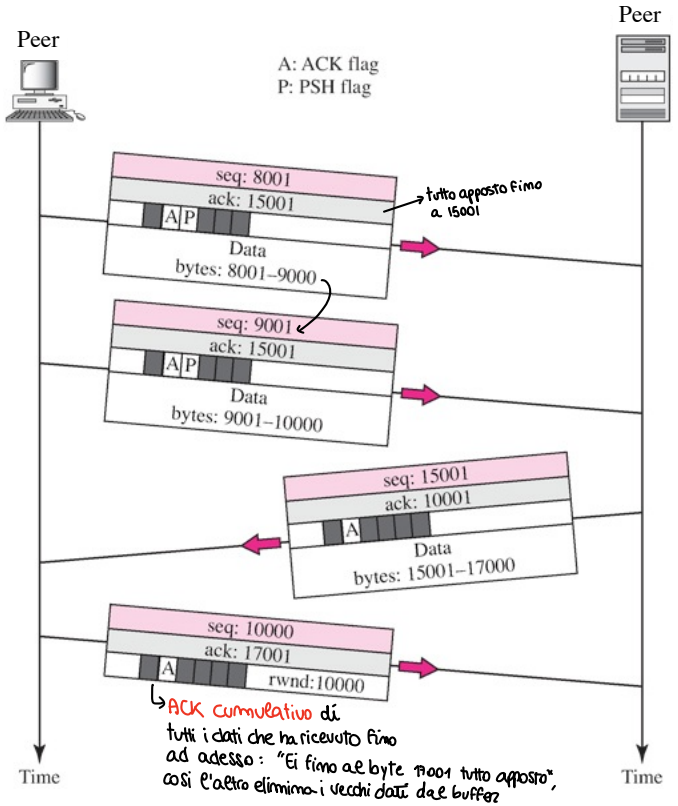
- A client does not know if a server is in passive opening - until he tries to connect. If it is not, he receives a RST segment
- Possibly (but very rare), also the other host may attempt an active opening at the same time (*simultaneous active opening*)
  - Both hosts send a SYN segment
  - the two segments cross each other
  - each host recognises this case because they receive a SYN after sending a SYN
  - in this case, both act as a server, answering with SYN+ACK, and then connection is established

↳ CASO in cui sono entrambi ATTIVI

State diagram of TCP - Phase 2: Data transfer

↳ questa fase può andare avanti all'infinito

- Both sides are in ESTABLISHED
- each segment carries data in one direction, and the acknowledge of the data in the opposite direction (*piggyback*)
- the two flows are independent



State diagram of TCP - Phase 2: Data transfer - Data PSH

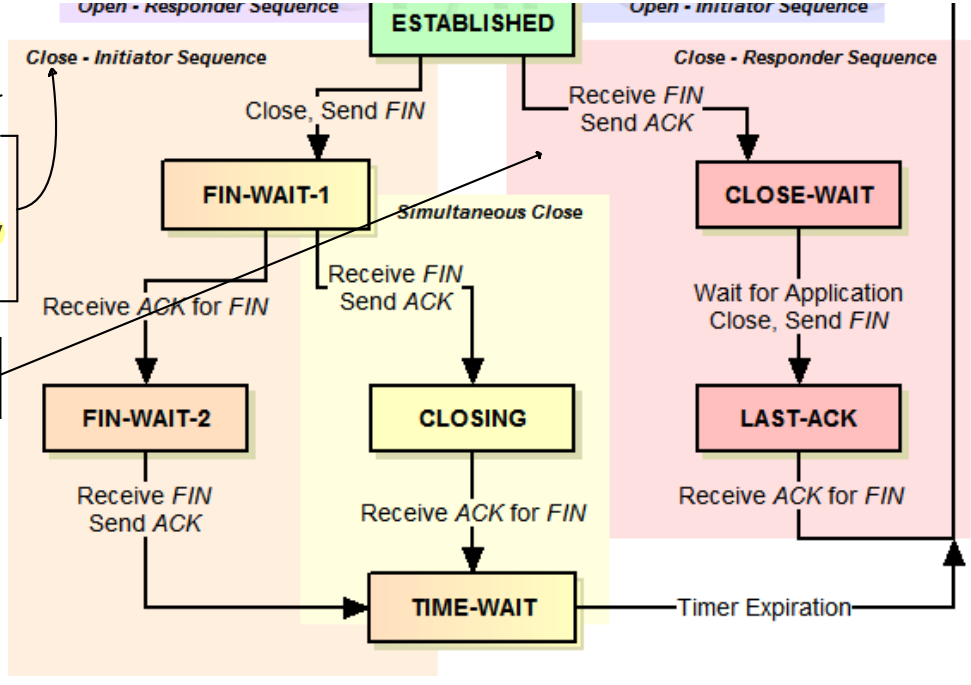
- TCP keeps data in buffers
  - outgoing: before assembling a segment
  - incoming: before passing it to applications
- This may yield latency and delays, which can be annoying in interactive applications
  ↳ es. SSH che digito i singoli caratteri che sto scrivendo sulla macchina host
- Sender application can ask for "data pushing", i.e., no buffering, as a special flag
  - if accepted, sender's TCP sends the data asap without buffering, activating the bit PSH
  - receiver's TCP finds PSH=1 and delivers the data as soon as possible (i.e. no buffering)

State diagram of TCP - Phase 2: Data transfer - URG data

- TCP is strictly sequential
- Sometimes it is useful to deliver data out of sequence (e.g. urgent signaling, abort commands)
- The application can ask for a urgent delivery during a send
  - if accepted, sender's TCP places urgent data at the beginning of the next segment, activates bit URG and sets urgent pointer accordingly
  - receiver TCP finds URG=1, separates urgent data from standard one, and delivers urgent data immediately, even if it is out of order

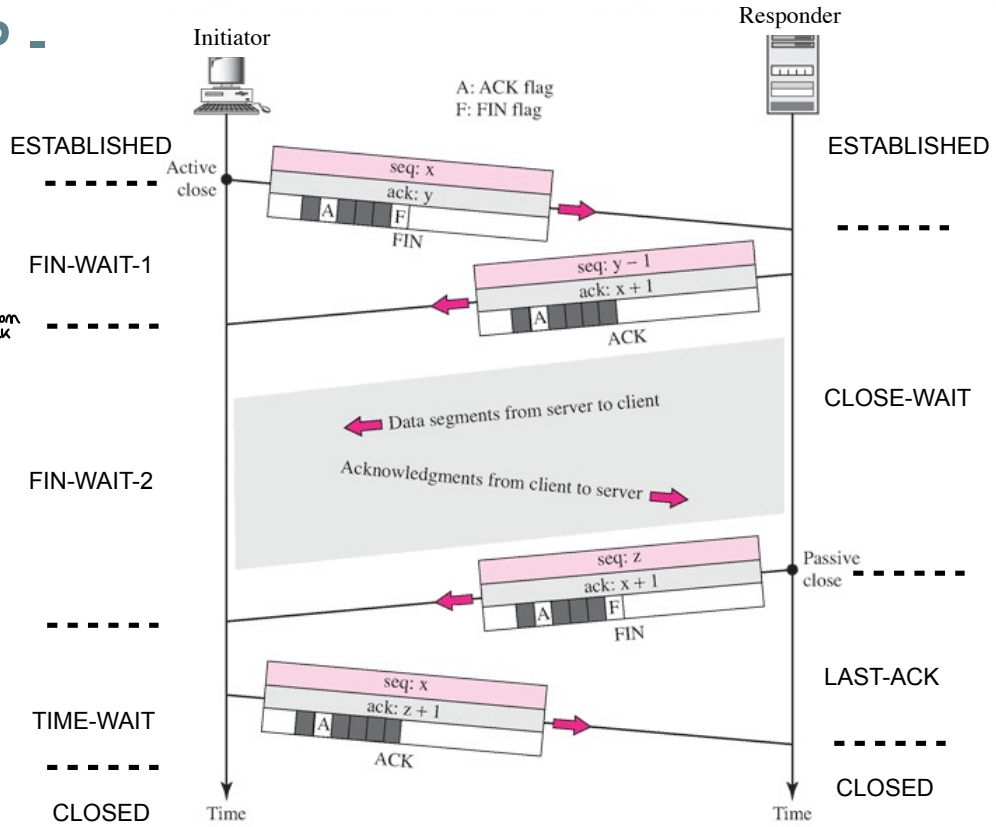
State diagram of TCP - Phase 3: Closing

- After some time, one of the two hosts may require to close the connection
- Closing is caused by "close" syscall on one of the two parties, which is called *the initiator* (not necessarily the initiator of Phase 1 - any party can start the closing sequence)
- The other party responds by executing the *responder* sequence.
- In the rare case that both parties start the initiator sequence at the same time, we have the *simultaneous close*



State diagram of TCP - Phase 3: Closing

- Host A initiates the closing procedure, by sending a segment with FIN=1, and moving to state FIN-WAIT-1
  - this segment can carry last data payload; no further new data can be sent by A after this → A ora può solo ricevere, non può inviare data, solo ACK
- Host B acknowledges this to A, sending ACK and moving to state CLOSE-WAIT; it notifies closure to application (i.e. end-of-file); A moves to FIN-WAIT-2
- Application on B can still send data to A, which will keep acknowledging it
- Eventually, application on B closes the socket; then B sends FIN to A and moves to state LAST-ACK
- A receives FIN, sends its ACK, and moves to TIME-WAIT
- B closes the socket when gets the ACK



TCP state diagram

- Why wait in TIME_WAIT? Two reasons:
  - the ACK sent after FIN_WAIT2 as reply to the FIN could be lost, and the server could re-send FIN - but then, if the client has cancelled all its informations, it receives a FIN which is not clear which connection it refers to
  - to avoid that old segments, still around in the network, can disturb a *reincarnation* of the same connection (i.e., the same ports and protocols are reused for another connection after that the previous one has terminated)

Se initiator mandasse un ACK e poi chiudesse subito, se l'ack si perdesse il responder rimanderebbe il FIN e initiator non lo riconoscerebbe

RESPONDER

INITIATOR

↳ stessa QUINTUPLA, ma nuova commessione

↳ TIMEOUT in cui NON posso riutilizzare per nuove commessioni le porte utilizzate nella scorsa commessione, in modo da far "esaurire" tutti i vecchi pacchetti in giro (o con TTL che scade o perché arrivano) ed evitare che nella nuova commessione arrivino vecchi pacchetti (che però non sono riconoscibili come vecchi perché possono avere stessa quintupla della nuova commessione)

netstat -na per vedere le socket attive, il loro stato e la dimensione dei buffer

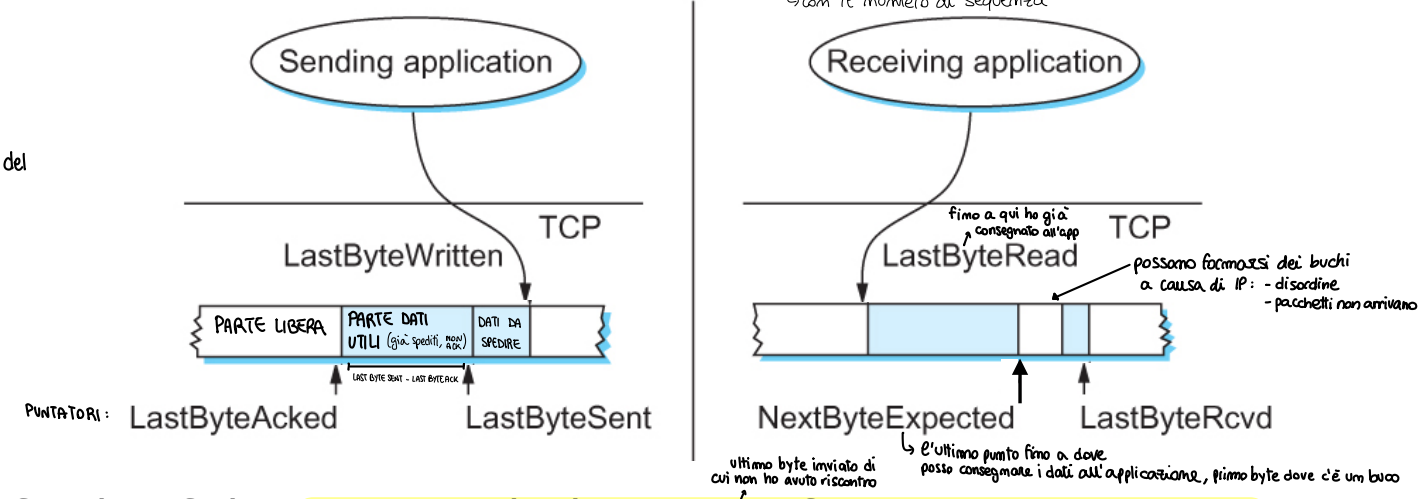
Sliding Window Revisited

- TCP's variant of the sliding window algorithm, which serves several purposes:
  1. it guarantees the reliable delivery of data,
  2. it ensures that data is delivered in order, → dato che si basa su IP, i pacchetti possono arrivare in disordine
  3. it enforces flow control between the sender and the receiver. → controllo di flusso END TO END

Sliding Window Revisited

Visto solo da una direzione, ma è bidirezionale
Risolve DUPLICATI, CONSEGNE FUORI ORDINE, RITRASMISSIONE e CONTROLLO di FLUSSO
↳ con il numero di sequenza

=> visione del SENDER



INVARIANTI

- Sending Side: LastByteAcked ≤ LastByteSent ≤ LastByteWritten
- Receiving Side: LastByteRead < NextByteExpected ≤ LastByteRcvd + 1
  ↳ un byte non può essere letto finché non è stato ricevuto ↳ potrebbero esserci dei buchi
- NextByteExpected is the value that the receiver sends to the sender in the Acknowledgment field of the TCP header

TCP Flow Control

→ come risolvo il CONTROLLO DI FLUSSO?

- In order to do not overrun the receiver's buffer, it must be
  LastByteRcvd − LastByteRead ≤ MaxRcvBuffer
  - ↳ il RICEVENTE controlla la velocità del MITTENTE comunicando una finestra che non sia più grande della quantità di dati da memorizzare
  - ↳ PARAMETRO che varia in base all'IMPLEMENTAZIONE dell'algo
- Hence, the receiver can still accept this amount of data:
  AdvertisedWindow = MaxRcvBuffer − ((NextByteExpected − 1) − LastByteRead)
  - ↳ quanti dati posso ancora ricevere → è inserito nel campo WINDOW SIZE dell'header del TCP e serve per far capire quanto spazio libero ha nel buffer durante la comunicazione
  - max 64 KB ← perché WINDOW SIZE ha dim. 16 bit → 2^{16} = 64 KB
- This is the value that the receiver sends to the sender in the *Window* field of the TCP header
- The sender, on the other side, must never send more data to the receiver than it can store in its buffer, that is:
  LastByteSent − LastByteAcked ≤ AdvertisedWindow
  - ↳ dati già inviati ma non ACK
- This includes the data already sent but not acknowledged yet, therefore the effective limit to the data we can still send is
  EffectiveWindow = AdvertisedWindow − (LastByteSent − LastByteAcked)
  - ↳ quanto può ricevere il mittente ora
  - ↳ quanto il Sender ha già mandato che non è stato "acknowledgeato"

Man mano che i dati arrivano il ricevitore li conferma e sposta verso dx LAST BYTERCVD, provocando una CONTRAZIONE della finestra. Il fatto che la finestra si contragga o no dipende da quanto velocemente l'applicazione utilizza i dati

RX : MAX − (NBE − 1 − LBR) = ADVERTISE
TX : (LBS − LBA) ≤ ADVERTISE

EFFECTIVE = ADVERTISED − (LBS − LA)
= MAX − (NBE − 1 − LBR) − (LBS − LA)

TCP Flow Control

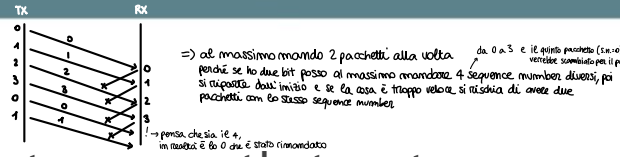
- Moreover, since the application sending process cannot overrun the sending buffer, it must be always:
  LastByteWritten − LastByteAcked ≤ MaxSendBuffer
  ↳ quanto sto occupando nel buffer di invio
- This means that if the sending process tries to write y bytes, but
  (LastByteWritten − LastByteAcked) + y > MaxSendBuffer
  then the "write" operation is blocked, until some space is freed in the sending buffer (i.e. until enough data is sent to the receiver and acknowledged by the receiver).
- In this way, the receiver process can slow down, or even stop, the transmission from sender process, by reducing the window to 0.

TCP Flow Control

- The sender can start transmitting again when the window opens again, that is, when the receiver process consumes some data.
- The new window can be notified by the receiver in the segments that it sends to the sender, if the receiver has any data to send to sender (the flow in the other direction)
  ↳ il RICEVITORE può anche avvisare con dei messaggi ACK (usando il TRASMITTER del suo host)
- But what if the receiver has no data to send to sender? The sender could never discover that the window is open!
- Thus, if the sender knows that AdvertisedWindow=0, it waits for a segment from receiver until a given timeout; if no segment is received, it sends a 1-byte "probe segment" to "stimulate" an ACK answer with the current window.
- This timeout will be increased exponentially between each successive probes (backoff algorithm)

Protecting against Wraparound

problema → in cui rischio di avere due pacchetti in rete con lo stesso sequence number



- As per Sliding Window protocol, we have to ensure that no two outstanding data (here bytes) carry the same sequence number
- The condition we saw was
  window width < maximum sequence number / 2
  equivalent to
  maximum sequence number > 2 × window width
  → #bit che ho a disposizione per il sequence number
  → larghezza della finestra
- In TCP: SequenceNum: 32 bits; AdvertisedWindow: 16 bits
- So the requirement of the sliding window algorithm is satisfied because $2^{32} \gg 2 \times 2^{16}$
- Recent implementations have more bits for AdvertisedWindow, but the condition still holds

Protecting against Wraparound

- Relevance of the 32-bit sequence number space (↳ si ripete ogni 4GB)
  - The sequence number used on a given connection might wraparound
  - A byte with sequence number x could be sent at one time, and then at a later time a second byte with the same sequence number x could be sent
  - Packets cannot survive in the Internet for longer than the **Maximum Segment Lifetime (MSL)** → è una stima di quanto tempo aspettare al max prima di dire che un pacchetto si è perso
  - MSL is set to 120 sec by default
  - We need to make sure that the sequence number does not wrap around within a 120-second period of time
  - Depends on how fast data can be transmitted over the Internet.

Bandwidth	Time until Wraparound
T1 (1.5 Mbps)	6.4 hours
Ethernet (10 Mbps)	57 minutes
T3 (45 Mbps)	13 minutes
Fast Ethernet (100 Mbps)	6 minutes
OC-3 (155 Mbps)	4 minutes
OC-12 (622 Mbps)	55 seconds
OC-48 (2.5 Gbps)	14 seconds

Keeping the Pipe Full

↳ quanto deve essere grande la finestra per sfruttare al massimo il canale

- AdvertisedWindow field must be big enough to allow the sender to keep the pipe full
  - (Clearly the receiver is free not to open the window as large as the AdvertisedWindow field allows)
- If the receiver has enough buffer space, the window needs to be opened far enough to allow a full *delay* × *bandwidth* worth of data

↳ RTT

↳ larghezza finestra per riempire il canale

Bandwidth	Delay × Bandwidth Product
T1 (1.5 Mbps)	18 KB → per riempire il canale devo avere dei buffer (finestra) da 18 KB
Ethernet (10 Mbps)	122 KB
T3 (45 Mbps)	549 KB
Fast Ethernet (100 Mbps)	1.2 MB
OC-3 (155 Mbps)	1.8 MB
OC-12 (622 Mbps)	7.4 MB
OC-48 (2.5 Gbps)	29.6 MB

Required window size for 100-ms RTT.

- A 16-bits field allows for 64kB window, which may be not enough for "long fat lines" (i.e., lines with *delay x bandwidth* >> 12500 bytes)
- TCP allows to set a *scaling factor* (up to 2^{14}) during the handshake protocol
  - Thus, the window can be up to 1GB ($2^{16} \times 2^{14} = 2^{30}$)

↳ ho 14 bit di scaling factor che ti dicono di quanto devi moltiplicare la grandezza della AdvertisedWindow, così non fa da collo di bottiglia

Triggering Transmission

- How does TCP decide to transmit a segment?
  - TCP supports a byte stream abstraction
    - ↳ senza confini il flusso
  - Application programs write bytes into streams
  - It is up to TCP to decide that it has enough bytes to send a segment → io produco 50 byte alla volta e me invio magari 200 byte alla volta (così ho meno overhead: spreco meno spazio per le intestazioni)
  - Ignore flow control: let us assume window is wide open, as would be the case when the connection starts

Triggering Transmission

i pacchetti TCP vengono incapsulati in pacchetti IP che a loro volta incapsulati in frame → MTU è la dim. massima

- TCP has three mechanism to trigger the transmission of a segment
  1. TCP maintains a variable Maximum Segment Size (MSS) and sends a segment as soon as it has collected MSS bytes from the sending process
     - MSS is usually set to the size of the largest segment TCP can send without causing local IP to fragment.
     - MSS: MTU of directly connected network – (TCP header + IP header)
       MTU → dim. max del frame; 20; 20 IPv4, 40 IPv6
  2. Sending process has explicitly asked TCP to send it → es. il PUSH
     - TCP supports push operation
  3. When a timer fires → se è lento a scrivere, invio i dati prima => NON EFFICIENTE, porta alla SILLY WINDOW SINDROME
     - Resulting segment contains as many bytes as are currently buffered for transmission (but always up to MSS and EffectiveWindow)

Silly Window Syndrome

- Basic TCP sliding window algorithm has no minimum size on transmitted segments.
- *Silly window syndrome* (SWS): a situation where many small, inefficient segments are sent (more overhead), rather than fewer large ones
- This can occur when a recipient advertises window sizes that are too small, or a transmitter is too aggressive in immediately sending out very small amounts of data.

↳ quando il ricevente comunica una finestra troppo piccola

- It is not a failure of the sliding window algorithm (which does its job to keep the receiving buffer full), but an inefficiency due to the network overhead

Silly Window Syndrome

- The worse case is when
  - AdvertisedWindow is 0
  - Receiver process consumes 1 byte
  - Then receiver sends to the sender that AdvertisedWindows is 1
  - Sender aggressively sends a 1-byte segment, closing the window
  - And repeat
- This leads to a sequence of 1-byte segments (each with 40 bytes of headers overhead)
- It would be better that sender waits a bit before sending a segment: the receiver process will consume more bytes, thus opening the window wider, and hence the sense could send a single larger segment

Nagle's Algorithm

- If there is data to send but the window is open less than MSS, then we may want to wait some amount of time before sending the available data
- But how long?
  - If we wait too long, we hurt interactive applications like Telnet and SSH
  - If we don't wait long enough, then we risk sending a bunch of tiny packets and falling into the silly window syndrome
- John Nagle introduced an elegant *self-clocking* solution
- Key Idea
  - As long as TCP has any data in flight, the sender will eventually receive an ACK
  - This ACK can be treated like a timer firing, triggering the transmission of more data

Nagle's Algorithm

When the application produces data to send: lo invio o no?

if both the available data and the window are ≥ MSS
    send a full segment
else (se non ho abbastanza dati da spedire per riempire un segmento)
    if there is unACKed data in flight
        buffer the new data until an ACK arrives → ACCUMULO
    else → se tutti i dati che ho inviato sono stati riscontrati
        send all the new data now

dati che ho inviato

Esercizio:

Una sorgente TCP ha inviato tre segmenti da 1200 byte l'uno, con SequenceNum pari a 3000, 4200, 5400 rispettivamente, e ha ricevuto i seguenti segmenti in questo ordine:

- ACK = 3000, AdvertisedWindow = 4200
- ACK = 5400, AdvertisedWindow = 1800
- ACK = 4200, AdvertisedWindow = 3000

Quanti byte può ancora spedire?

L'ACK cumulativo con SequenceNum maggiore ha SequenceNum = 5400 e AdvertisedWindow = 1800. La sorgente ha inviato anche i byte con SequenceNum 5399-6599 per cui non abbiamo ancora ricevuto l'ACK, per cui possiamo ancora spedire

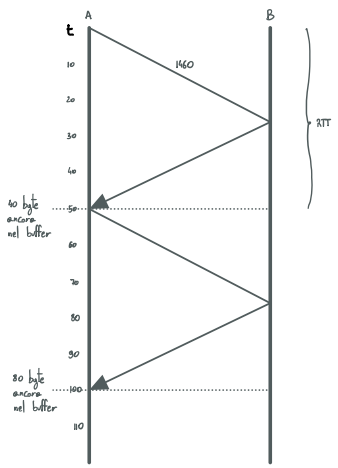
EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)
= 1800 - (6599 - 5399) = 600 Byte

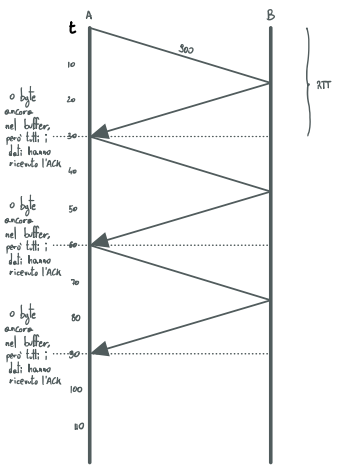
Esercizio:

Un'applicazione sta scrivendo su una socket TCP una stringa di 300 byte ogni 10 ms. La connessione TCP ha un MSS di 1460 byte e un RTT di 50 ms. Si supponga che CongestionWindow e AdvertisedWindow siano sufficientemente grandi.

A. Quanto è grande il payload di ogni segmento inviato dall'host, in media?
B. Cosa succede se il RTT scende a 30 ms?

A. Devo applicare l'algoritmo di Nagle. Siccome l'applicazione scrive 300 byte ogni 10 ms e l'RTT è di 50 ms, l'applicazione riesce a scrivere 1500 Byte durante un RTT. Ma MSS < di 1500 Byte, quindi l'applicazione invierà segmenti con 1460 Byte di payload.
B. Se il RTT scende a 30 ms, l'applicazione riuscirà a scrivere 900 Byte durante ogni MSS. Quindi il payload del singolo segmento, che verrà inviato alla ricezione di ogni ACK, sarà di 900 Byte.





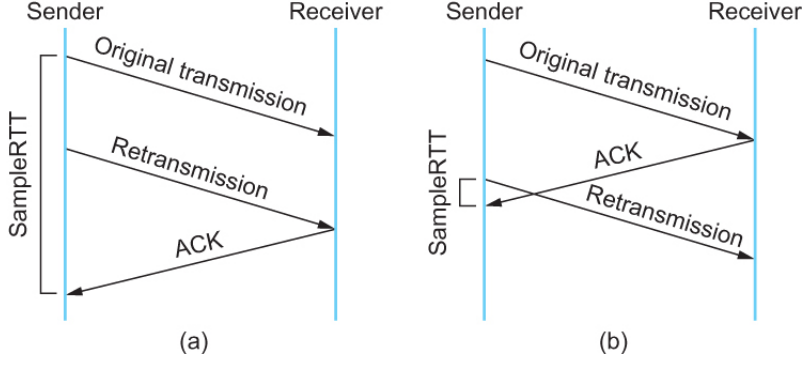
Adaptive Retransmission

- A segment must be retransmitted if we do not receive an acknowledge for some time
- How to set the timeout? We must adapt to the delays → i ritardi non sono fissi, ⇒ RTT è variabile
- Original Algorithm
  - Measure SampleRTT for each segment/ ACK pair
  - Compute weighted average of RTT
    - EstRTT := α × EstRTT + (1 - α) × SampleRTT
      (α × EstRTT: stima precedente; (1 - α) × SampleRTT: nuovo campione)
    - α between 0.8 and 0.9
  - Set timeout based on EstRTT
    - TimeOut = 2 × EstRTT → poi non sarà questo

Timeout calculation - Original Algorithm

- Problem:
  - ACK does not really acknowledge a transmission, but only the receipt of data
  - When a segment is retransmitted and then an ACK arrives at the sender, it is impossible to decide if this ACK should be associated with the first (a) or the second (b) transmission for calculating RTTs

↳ considero solo i pacchetti che non vengono ritrasmessi, per il calcolo del RTT stimato



Karn-Partridge Algorithm

RTT varia molto a causa delle code dei router intermedi, il tempo di attraversamento di un router cambia (mom è fisso)

- Solution:
  - Do not sample RTT when retransmitting — perché non so dire se un ACK si riferì alla prima trasmissione o alla seconda
  - Double timeout after each retransmission
- Karn-Partridge algorithm is an improvement over the original approach, but we still need to understand how timeout is related to congestion
  - If you timeout too soon, you may unnecessarily retransmit a segment which adds load to the network
- Main problem with the original computation is that it does not take variance of Sample RTTs into consideration.
- If the variance among Sample RTTs is small → se la varianza è piccola, vuol dire che i router lavorano in maniera stabile
  - Then the Estimated RTT can be better trusted
  - There is no need to multiply this by 2 to compute the timeout
- On the other hand, a large variance in the samples suggest that timeout value should not be tightly coupled to the Estimated RTT
  - ↳ vuol dire che la mia situaz. sta cambiando (in bene o in male) e non è più veritiero il dato

Jacobson-Karels Algorithm

- Jacobson and Karels proposed a new scheme for TCP retransmission
- Set Difference = SampleRTT − EstimatedRTT (RTT ULTIMO ACK; positivo: RTT aumenta; negativo: RTT diminuisce)
- EstimatedRTT := EstimatedRTT + (δ × Difference) (quanto PESA la differenza)
- Deviation := Deviation + δ × (|Difference| − Deviation) (media della differenza)
  - δ = 1 - α, is a fraction between 0 and 1
- TimeOut = μ × EstimatedRTT + Φ × Deviation (Φ = 4)
  - where based on experience, μ is typically set to 1 and Φ is set to 4.
- Thus, when the variance is small, TimeOut is close to EstimatedRTT; a large variance causes the deviation term to dominate the calculation.

TCP: Performance

- Two metrics: *latency* and *throughput*
- Experimental Setup: Xeon 2.4 GHz, dual Gigabit Ethernet in link aggregation, Linux
  - datalink channel limit: almost 2Gbps full duplex, with no losses
- The larger the MSS the higher the throughput, but above 1KB does not increases substantially
  - Typical MSS = 1460 byte (suited when interface has MTU=1500)
- keeps below 2Gbps (several bottlenecks)
- overall, TCP can reach high throughputs
- Sometimes the bottleneck is the memory itself