Computer Networks
Chapter 6 - Congestion Control and Resource Allocation

Prof. Marino Miculan

Problem

- We have seen enough layers of protocol hierarchy to understand how data can be transferred among processes across heterogeneous networks

- Problem
  - How to **effectively** and **fairly** allocate **resources** among a collection of competing users?

sono usate contemporaneamente da miliardi di utenti

risorse della rete, ovvero canali, banda, dispositivi e la loro memoria interna

Chapter Outline

- Issues in Resource Allocation
- Queuing Disciplines
- TCP Congestion Control
- Congestion Avoidance Mechanism

Congestion Control and Resource Allocation

PARTIAMO in generale

- Resources in networks:
  - Bandwidth of the links
  - Buffers at the routers and switches
- Packets contend at a router for the use of a link, with each packet placed in a queue waiting for its turn to be transmitted over the link
  ↳ nella memoria di router/switch. Non essendo illimitata, se è piena allora si butta via tutto quello che arriva → CONGESTIONE
- When too many packets are contending for the same link
  - The queue overflows
  - Packets get dropped
  - **Network is congested!**
- Network should provide a **congestion control / avoidance mechanism** to deal with such a situation → non possiamo controllare i singoli router perché non sappiamo che percorso farà

Congestion Control and Resource Allocation

- Congestion control and Resource Allocation are two sides of the same coin
- If the network takes active role in **allocating resources**
  - The congestion may be avoided
  - No need for congestion control
- But allocating beforehand resources with any precision is difficult → difficile prevedere all'inizio il numero di risorse che ci servirannno
  - Resources are distributed throughout the network
  - Not easily definable beforehand
- On the other hand, we can always let the sources send as much data as they want then recover from the congestion when it occurs (**congestion control**)
  ↳ permetto che succeda congestione, però me ne devo accorgere e avvio → CURARE
  - Easier approach but it can be disruptive because many packets many be discarded by the network before congestions can be controlled
- A third approach is to let the sources send as much data as they want but stopping them *before* congestion is going to occur (**congestion avoidance**)
  ↳ cercare di accorgersi quando sta per avvenire senza comunicare all'inizio quante risorse servono → PREVENIRE
  - More difficult (must recognise congestion before it happens), less implemented, and usually together with congestion control

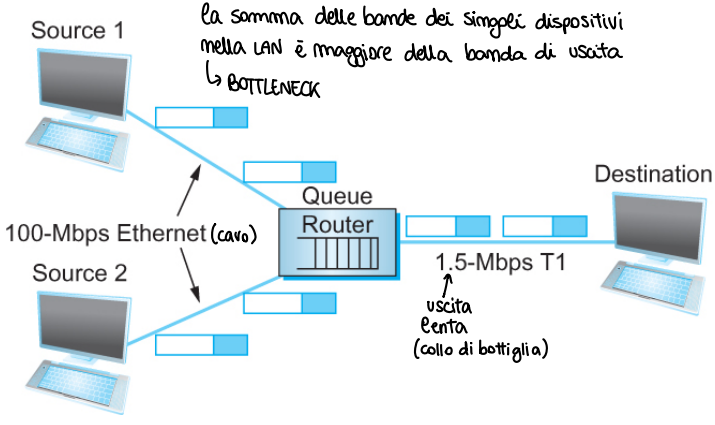
Congestion Control and Resource Allocation

dove vengono implementati i controlli → nei dispositivi / negli host (estremi)] non sono esclusivi

- Congestion control and resource allocations involve both hosts and network elements such as routers
- In **network elements**
  - Various queuing disciplines can be used to control the order in which packets get transmitted and which packets get dropped
- At the **hosts' end**
  - The congestion control mechanism paces how fast sources are allowed to send packets

Issues in Resource Allocation

- Network Model: Packet Switched Network → ogni nodo è attraversato da più pacchetti
  - We consider resource allocation in a packet-switched network (or internet) consisting of multiple links and switches (or routers).
  - In such an environment, a given source may have more than enough capacity on the immediate outgoing link to send a packet, but somewhere in the middle of a network, its packets encounter a link that is being used by many different traffic sources



A potential bottleneck router.

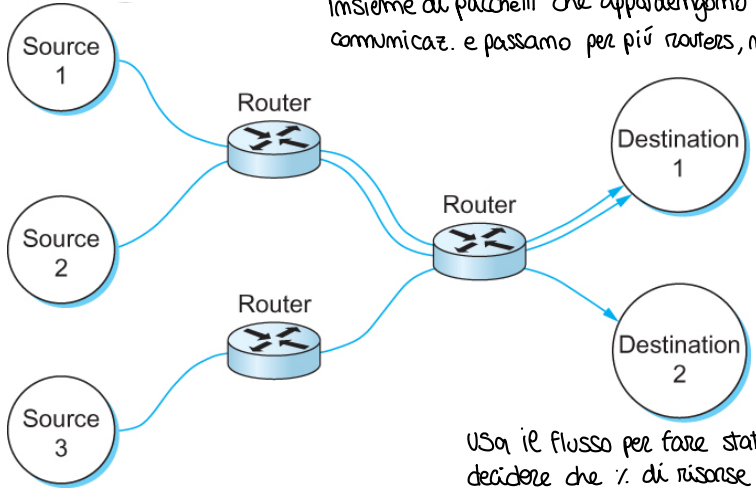
Issues in Resource Allocation

- Network Model: **Connectionless Flows**
  - For much of our discussion, we assume that the network is essentially *connectionless*, with any connection-oriented service implemented at transport level (e.g. TCP over IP).
  - We need to qualify the term "connectionless" because our classification of networks as being either connectionless or connection-oriented is a bit too restrictive
  - In particular, the assumption that all datagrams are completely independent in a connectionless network is too strong.
  - The datagrams are certainly switched independently, but it is usually the case that a stream of datagrams between a particular pair of hosts flows through a particular set of routers

Issues in Resource Allocation

- Network Model: **Connectionless Flows**

insieme di pacchetti che appartengono alla stessa comunicaz. e passano per più routers, ma fanno lo stesso percorso



Usa il flusso per fare statistiche e decidere che % di risorse assegnare al flusso

→ nei pacchetti IPv6 per aiutare il router a capire il flusso di ciascun pacchetto ha nell' header FLOW LABEL

Multiple flows passing through a set of routers

Issues in Resource Allocation

- Network Model: **Connectionless Flows**
  - One of the powers of the flow abstraction is that flows can be defined at different granularities.
  - A flow can be **host-to-host** (i.e., have the same source/destination host addresses) or **process-to-process** (i.e., have the same source/destination host/port pairs).
  - In the latter case, a flow is essentially the same as a **channel**, as we have been using that term so far.
    The reason we introduce a new term is that **a flow is visible to the routers inside the network** (i.e. at network level), whereas a channel is an end-to-end abstraction.

Issues in Resource Allocation

- Network Model: **Connectionless Flows**
  - Because multiple related packets flow through each router, it sometimes makes sense to maintain some **state information for each flow**, e.g. rate, size of packets, bitrate, etc.
  - This information can be used to make resource allocation decisions about the packets that belong to the flow. This state is sometimes called **soft state**. → i router dovrebbero essere stateless, ovvero non ricordare nulla del passato, in realtà imparo dal traffico che attraversa (ma senza un automa)
  - The main difference between soft state and "hard" state is that soft state need not always be explicitly created and removed by signalling.
  - Often it is inferred "automagically" by the routers (recent techniques adopt deep learning on traffic data)

↳ es. identifico i pacchetti per IP-PORTA (può causare problemi con il NAT in quanto flussi differenti vengono mappati sulla stessa socket (IP-PORTA)
oppure i router usano il MACHINE LEARNING per imparare

Issues in Resource Allocation

- Network Model: **Connectionless Flows**
  - Soft state represents a middle ground between a purely connectionless network that maintains no state at the routers and a purely connection-oriented network that maintains hard state at the routers.
  - The correct operation of the network does not depend on soft state being present (each packet is still routed correctly without regard to this state), but when a packet happens to belong to a flow for which the router is currently maintaining soft state, then the router is better able to handle the packet.

Issues in Resource Allocation

- Taxonomy: Router-centric versus Host-centric
  - In a **router-centric design**, each router takes responsibility for deciding when packets are forwarded and selecting which packets are to dropped, as well as for informing the hosts that are generating the network traffic how many packets they are allowed to send.
  - In a **host-centric design**, the end hosts observe the network conditions (e.g., how many packets they are successfully getting through the network) and adjust their behavior accordingly.
  - Note that these two approaches are not mutually exclusive.

Issues in Resource Allocation

- Taxonomy: Reservation-based versus Feedback-based
  - In a **reservation-based system**, some entity (e.g., the end host) asks the network for a certain amount of capacity to be allocated for a flow. → funziona bene dentro un AS
    - Each router then allocates enough resources (buffers and/or percentage of the link's bandwidth) to satisfy this request. If the request cannot be satisfied at some router, because doing so would overcommit its resources, then the router rejects the reservation.
  - In a **feedback-based approach**, the end hosts begin sending data without first reserving any capacity and then adjust their sending rate according to the feedback they receive. ↳ richiede fiducia, a volte i router "concorrenti" possono fingere di essere congestionati a pacchetti di aziende rivali
    - This feedback can either be explicit (i.e., a congested router sends a "please slow down" message to the host) or it can be implicit (i.e., the end host adjusts its sending rate according to the externally observable behavior of the network, such as packet losses).

es. reale: Voglio andare a Roma in hotel e li ho il parcheggio libero, ma in autostrada c'è coda tra poco. Io lo so perché guardo i cartelloni che avvisano → FEEDBACK esplicito

Issues in Resource Allocation

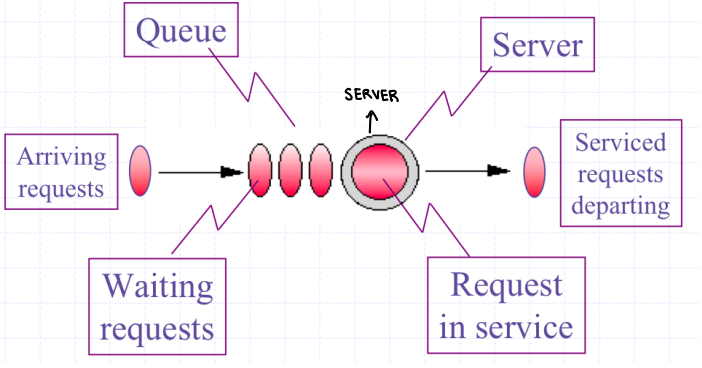
- Evaluation Criteria: **Effective Resource Allocation**
  - Effectiveness of a resource allocation scheme can be measured by considering the two principal metrics of networking: **throughput** and **delay.**
  - We want as much throughput and as little delay as possible, but these goals are at odds with each other
  - We can increase throughput by allowing as many packets into the network as possible, so as to drive the utilisation of all the links up to 100%.
    ↳ utilizzo massimo se riempio tutti i LINK al 100%.
  - But **increasing the number of packets in the network also increases the length of the queues at each router, and hence leads to longer delays**
  - On the converse, **delays can be minimised by keeping queues as empty as possible, but this means lower utilisation of links**

Issues in Resource Allocation

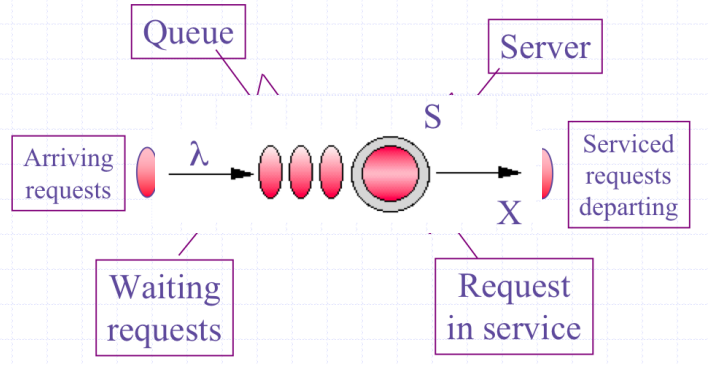
- **Effective Resource Allocation**

  TEORIA DELLE CODE

  - A detailed analysis would require tools and results from an area of Probability and Statistics called **Queue Theory**
  - As an example, let us consider a completely random flow of requests to one Memoryless server (this kind of queue is called M/M/1)



Issues in Resource Allocation



- Parameters:
  - Average arrival rate into the queue: λ [req/s]. → valore medio
  - Average service time at the server: S [s/req].
    ↳ secondi per ogni richiesta
- Results:
  - Average residence time a customer spends getting through the queueing system (i.e., **average total delay**): **R = S / (1 − λS)**.
    ↳ RITARDO CODA + SERVIZIO
  - Utilization of the server: **ρ = λS**. va sempre < 1, se =1 la coda diventa ∞ — ∈ [0, 1]
  - Average queue length of the system: **Q = λR**.
  - Average waiting time a customer can expect to spend before getting service: **W = R — S = Sρ/(1 — ρ)**.
- (There are many tools and libraries for these simulations and analysis)

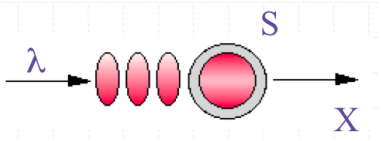
Issues in Resource Allocation

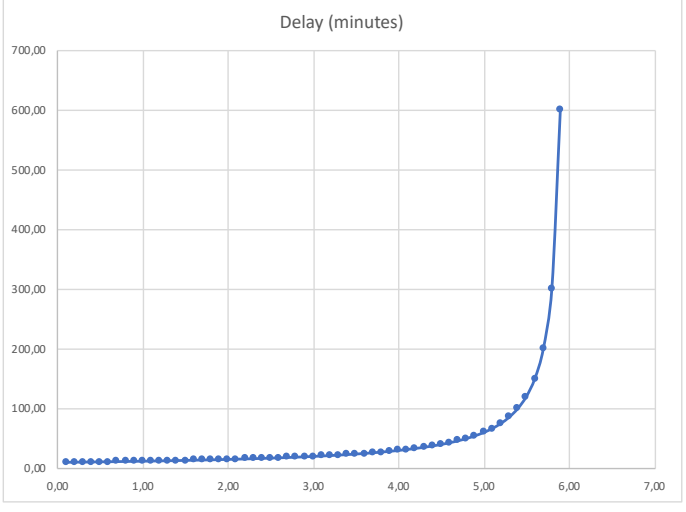
- Evaluation Criteria: Effective Resource Allocation
  - To describe this relationship, some designers have proposed to use **the ratio of throughput to delay** as a metric for evaluating the effectiveness of a resource allocation scheme
  - This ratio is sometimes referred to as the **power** of the network.

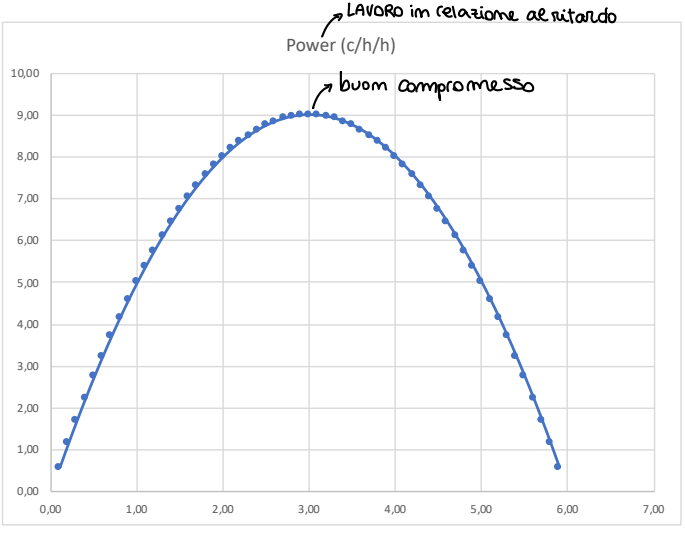
    Power = Throughput/Delay
  - If we aim to both **maximize throughput** and **minimise delay**, we aim to **maximise power**.
    - (But in some contexts one of the two aspects is more important, and must be prioritised)
  - In the M/M/1 case of the previous slide: **Power = $\lambda/S - \lambda^2$**

Issues in Resource Allocation

- Example: a simple bank office
  - One teller with S=10 min/client
  - λ range from 0.1 to 5.9 clients/hour

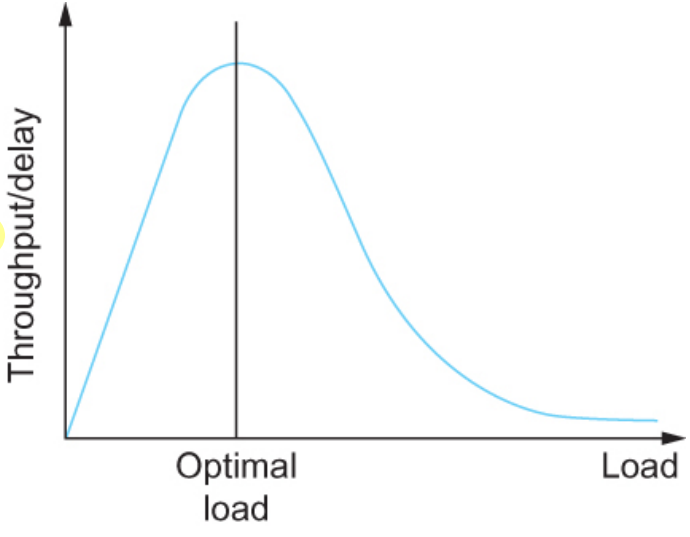






Issues in Resource Allocation

- In real settings, the power follows a curve like the one aside
- At low loads (low λ), queues are almost empty, so power increases almost linearly
- At a certain point, power reaches a maximum. Throughput is well below the capacity of the channels but queues are still short
- After this point, throughput increases but packets start rapidly to queue up, increasing the delay (and decreasing the power)
- Ideally we should aim to keep the system around the optimal point



Power as a function of load

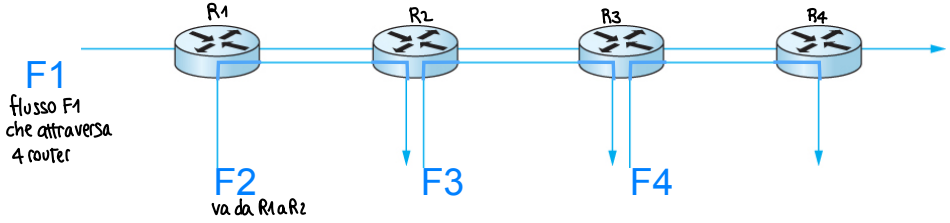
Issues in Resource Allocation

- Evaluation Criteria: **Fair Resource Allocation** (dispositivi, router)
  - The effective utilization of network resources is not the only criterion for judging a resource allocation scheme.
  - We must also consider the issue of fairness. However, we quickly get into murky waters when we try to define what exactly constitutes fair resource allocation.
  - For example, a reservation-based resource allocation scheme provides an explicit way to create controlled unfairness.
  - With such a scheme, we might use reservations to enable a video stream to receive 1 Mbps across some link while a file transfer receives only 10 Kbps over the same link.

Issues in Resource Allocation

- Evaluation Criteria: **Fair Resource Allocation**
  - In the absence of explicit information to the contrary, when several flows share a particular link, we would like for each flow to receive an equal share of the bandwidth.
  - This definition presumes that a fair share of bandwidth means an equal share of bandwidth.
  - But even in the absence of reservations, **equal shares may not equate to fair shares**.
  - Should we also consider the length of the paths being compared?
    - Consider the figure in next slide.

Issues in Resource Allocation



- One four-hop flow F1 and three two-hop flows F2,F3,F4
- Let B be the overall band of each router. Overall, the resources to be assigned are 4B ⟶ ogni flusso dovrebbe avere assegnato B banda
- If each router assigns equal parts of its band to each incoming flow:
  - F1 gets B/2 (R1) + B/3 (R2) + B/3 (R3) + B/2 (R4) = 5B/3 (F1 prende il doppio di F2)
  - F2 gets B/2 + B/3 = 5B/6
  - F3 gets B/3 + B/3 = 2B/3
  - F4 gets B/3 + B/2 = 5B/6

=> F1 ha più banda degli altri, non è FAIR anche se i router assegnano B in modo fair (rispetto ai flussi che li percorrono)

- hence F1 gets way more than the others!

dare fette uguali non significa essere equi

Issues in Resource Allocation

- Evaluation Criteria: **Fair Resource Allocation**
  - Assuming that fair implies equal and that all paths are of equal length, **Jain's fairness index** can be used to quantify the fairness of a congestion-control mechanism.
  - given a set of flow throughputs x1, x2, … , xn (measured in e.g., bps), Jain's fairness index f is defined as follows:

$$\frac{1}{n} \leq f(x_1, x_2, \ldots, x_n) = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \sum_{i=1}^{n} x_i^2} \leq 1$$

es. per 4 flussi:

$$f(x_1, \ldots, x_4) = \frac{(x_1 + x_2 + x_3 + x_4)^2}{4(x_1^2 + x_2^2 + x_3^2 + x_4^2)}$$

  - Can be applied to other resources as well (e.g. memory, power, etc.)

Se $x_1 \neq 0$, $x_i = 0 \ \forall i \neq 1$ (gli altri hanno 0 bps di banda assegnata) $f(x_1, \ldots, x_n) = \frac{1}{n}$ => situazione più UNFAIR (MINIMA FAIRNESS)

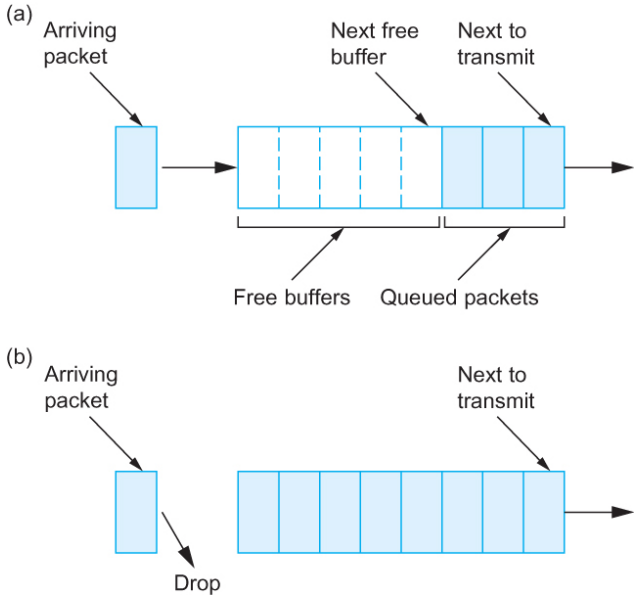
Issues in Resource Allocation

- Evaluation Criteria
  - The fairness index always results in a number between 1/n and 1 – the greater, the better
    - 1/n when all x_i but one are 0, i.e., when all the resources go to one flow => lowest fairness
    - 1 when $x_i = x_j$ for all i, j => greatest fairness.
  - In the example of four flows as above:
    f(5B/3, 5B/6, 2B/3, 5B/6) = 0.867

Queuing Disciplines: FIFO with tail drop

- The idea of FIFO queuing, also called first-come-first-served (FCFS) queuing, is simple:
  - The first packet that arrives at a router is the first packet to be transmitted
  - Given that the buffer space at each router is finite, if a **packet arrives and the queue (buffer space) is full, then the router discards that packet**
    ↳ Es fanno tutti i router/switch economici
  - This is done without regard to which flow the packet belongs to or how important the packet is. This is sometimes called **tail drop**, since packets that arrive at the tail end of the FIFO are dropped
  - Note that tail drop and FIFO are two separable ideas:
    FIFO is a scheduling discipline—it determines the order in which packets are transmitted.
    Tail drop is a drop policy—it determines which packets get dropped

Queuing Disciplines: FIFO



(a) FIFO queuing; (b) tail drop at a FIFO queue.

Queuing Disciplines: Priority

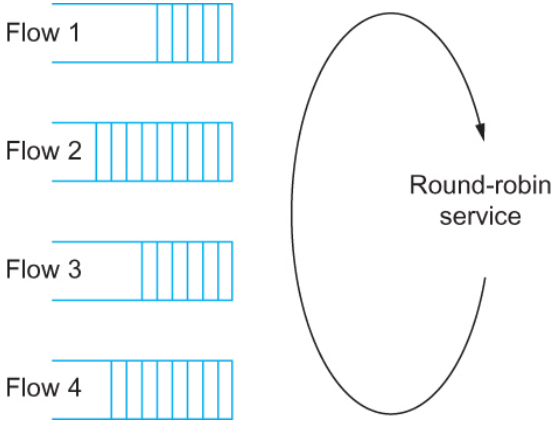
è meglio tenere code separate in base alla priorità

- A simple variation on basic FIFO queuing is **priority queuing**.
- The idea is to mark each packet with a priority
  - the mark could be carried, for example, in the IP header (TOS/DSCP), or determined in other ways
- The routers then implement multiple FIFO queues, one for each priority class. The router always transmits packets out of the highest-priority queue if that queue is nonempty before moving on to the next priority queue.
- Within each priority, packets are still managed in a FIFO manner.

Queuing Disciplines: Fair Queuing

→ spesso i provider quando ti lamenti ti "danno più banda"

- The main problem with FIFO queuing is that it does not discriminate between different traffic sources, or it does not separate packets according to the **flow** to which they belong.
- **Fair queuing (FQ)** is an algorithm that has been proposed to address this problem. → un'approssimazione del ROUND ROBIN byte a byte
- The idea of FQ is to maintain a separate queue for each flow currently being handled by the router. The router then services these queues in a sort of round-robin



(Not really) Round-robin service of four flows at a router

se io servo in questo modo 1234 1234 1234 ... → allora è equo
però se ho pacchetti di dimensione diversa non posso interrompere durante la trasmissione di un pacchetto

Queuing Disciplines: Fair Queuing

- The main complication with Fair Queuing is that the packets being processed at a router are not necessarily the same length.
- To truly allocate the bandwidth of the outgoing link in a fair manner, it is necessary to take packet length into consideration.
  - For example, if a router is managing four flows, one with 1000-byte packets and the others with 100-byte packets, then a simple round-robin servicing of packets from each flow's queue will give the first flow 1000/1300 = 77% of the link's bandwidth and only 100/1300 = 7.7% of its bandwidth to each other flow
  - The Jain's index would be f(1000,100,100,100) = 0.41, very close to the minimum (0.25)

Queuing Disciplines: Fair Queuing

- What we really would like to have is bit-by-bit round-robin; that is, the router transmits a bit from flow 1, then a bit from flow 2, and so on.
- Clearly, it is not feasible to interleave the bits from different packets. Each packet must be transmitted entirely.
- The FQ mechanism therefore approximates this behavior by first determining when a given packet would finish being transmitted if it were being sent using bit-by-bit round-robin, and then using this finishing time to sequence the packets for transmission.

Queuing Disciplines: Fair Queuing

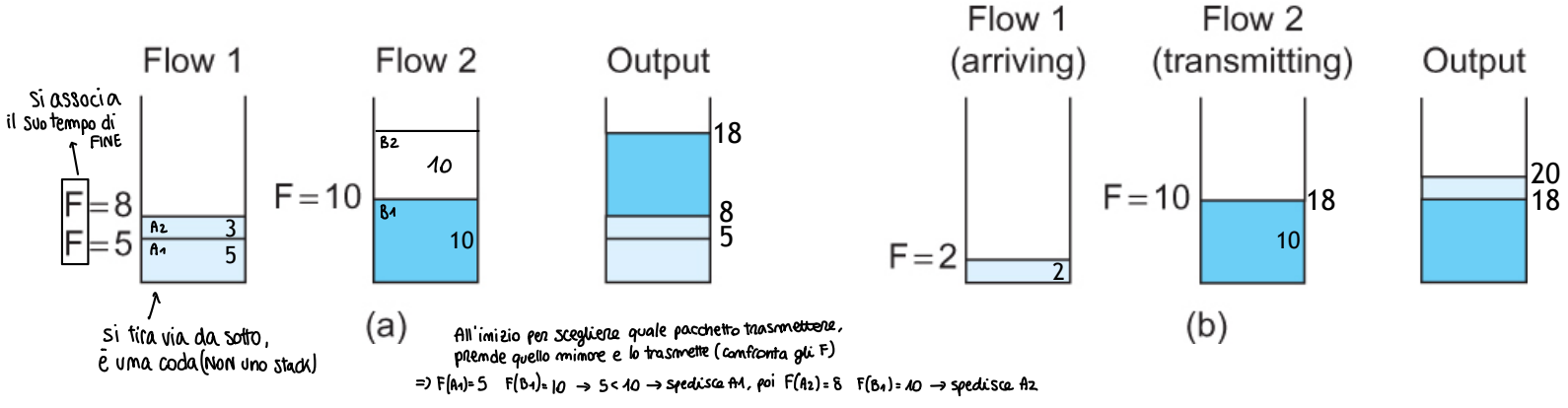
- To understand the algorithm for approximating bit-by-bit round robin, consider the behavior of a single flow
- For this flow, let
  - P_i : denote the length of packet i (measured in time)
  - S_i: time when the router starts to transmit packet i
  - F_i: time when router finishes transmitting packet I
  - $F_i = S_i + P_i$

  ↳ LO STIMI all'inizio, non mano a mano
- When do we start transmitting packet i?
  - Depends on whether packet i arrived before or after the router finishes transmitting packet i-1 for the flow
- Let A_i denote the time that packet i arrives at the router
- Then the start time is $S_i = \max(F_{i-1}, A_i)$
- Hence $F_i = \underbrace{\max(F_{i-1}, A_i)}_{S_i} + P_i$

Queuing Disciplines: Fair Queuing

- Algorithm: For each flow, we calculate a timestamp F_i for each packet that arrives as follows
  - We have F_{i-1}, the timestamp of last packet, and arrival time A_i and length P_i
  - Then $F_i = \max(F_{i-1}, A_i) + P_i$
- Next packet to transmit is always the packet that has the lowest timestamp, among all flows
  - I.e., the packet that should finish transmission before all others
- This guarantees minimum differences between flows, hence highest fairness

Queuing Disciplines: Fair Queuing



Example of fair queuing in action:
(a) packets with earlier finishing times are sent first;
(b) sending of a packet already in progress is completed

=> mentre sta trasmettendo un pacchetto NON può INTERROMPERE la trasmissione per qualcun altro! Bisogna attendere la fine e poi si sceglie 1 pacchetto da spedire tra i vari flussi

Queuing Disciplines: Fair Queuing

- Let us consider a router managing four flows
  - F1: 1000-byte packets
  - F2,F3,F4: 100-byte packets
- According to Fair Queuing:
  - first we transmit one packet from F2, one from F3, one from F4, one from F2, etc. until 10 packets from each flow are sent
  - Then we can transmit one packet from F1
  - and then we repeat
- The band given to each flow is the same
- Fairness = 1

ogni volta scelgo il pacchetto che ha Fi minore rispetto a quelli degli altri flussi

→ ruoto in ROUND ROBIN tra F2, F3, F4

es.

	A		B		C
14	A_3				
12	A_2	15	B_2	15	C_2
5	A_1	4	B_1	9	C_1

C_2 B_2 A_3 A_2 C_1 A_1 B_1 →
44 38 27 25 18 9 4

Example

Un router applica l'accodamento FairQueing a tre flussi A,B,C, con attualmente pacchetti della seguente durata (in qualche unità di tempo): A: 100,200,100,1000; B: 300,200,500; C: 400. (a) A che istante termina la trasmissione del pacchetto di C? (b) Se all'istante 1200 arriva un altro pacchetto del flusso C di durata 100, quando inizia la sua trasmissione?

Packet	Duration	Finish time
A1	100	100
A2	200	300
A3	100	400
A4	1000	1400

Packet	Duration	Finish time
B1	300	300
B2	200	500
B3	500	1000

Packet	Duration	Finish time
C1	400	400
C2	100	1300

Output order

Packet	Start	Finish time
A1	0	100
A2	100	300
B1	300	600
C1	600	1000
A3	1000	1100
B2	1100	1300
B3	1300	1800
C2	1800	1900
A4	1900	2900

ogni volta prendo il pacchetto (da uno dei 3 flussi) che ha finish time minore
↳ li calcolo all'inizio, non mano a mano quindi non è del tutto efficiente

Answers:
(a) 1000
(b) 1800

TCP Congestion Control

- TCP congestion control was introduced into the Internet in the late 1980s by Van Jacobson, roughly eight years after the TCP/IP protocol stack had become operational. → controllo della congestione è obbligatorio
- Before this time, the Internet was suffering from *congestion collapse*
  - 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 Congestion Control

- The idea of TCP congestion control is that each source has to determine how much capacity is available in the network, so that it knows how many packets it can safely have in transit.
  - Nagle's algorithm: Once a given source has this many packets in transit, it uses the arrival of an ACK as a signal that one of its packets has left the network, and that it is therefore safe to insert a new packet into the network without adding to the level of congestion.
  - By using ACKs to pace the transmission of packets, TCP is said to be self-clocking.

TCP Congestion Control

- Since the late ‘80s, many congestion control algorithms have been (and still are) proposed for TCP
  - The Wikipedia web page lists more than 30 algorithms
- They differ on
  - The feedback observed for the control
  - The nodes involved in the algorithm (sender / receiver / routers)
  - The parameters it aims to optimise (latency / bandwidth / retransmits…)
  - Fairness
- Most of these algorithms can coexist on the same network
  - We can have different cong. contr. algorithms at different nodes

TCP Congestion Control

- Every Operating System has a "general purpose" algorithm by default:
  - **CUBIC**: default in Linux kernels since version 2.6.19. (Nov. 2006)
  - **Compound TCP**: default in Windows since Vista and Server 2008
- The administrator can change the default algorithm with others, if needed (with great care!). E.g., more recent algorithms include
  - **Proportional Rate Reduction (PRR)**: available in Linux kernels to improve loss recovery since version 3.2. (January 2012)
  - **Bottleneck Bandwidth and Round trip-time (BBR)**: Developed by Google for QUIC (2016) available in Linux kernels to enable model-based congestion control since version 4.9. Not based on packet loss.
- We will see only a few ingredients, adopted in classical algorithms (*Tahoe* 1988, and *Reno* 1990) but still used nowadays

TCP Congestion Control

- **Congestion Window** (assieme ad Advertised Window, ecc.) → quanti pacchetti posso ancora mandare prima di congestionare
  - TCP maintains a new state variable for each connection, called **CongestionWindow** or **cwnd**, which is used by the source to limit how much data it is allowed to have in transit at a given time.
  - The congestion window is congestion control's counterpart to flow control's advertised window.
  - TCP is modified such that the maximum number of bytes of unacknowledged data allowed is now the **minimum of the congestion window and the advertised window**

TCP Congestion Control

- **Effective Window** ↳ sliding window modificata
  - TCP's effective window is revised as follows:
    - **MaxWindow = MIN(CongestionWindow, AdvertisedWindow)**
    - EffectiveWindow = MaxWindow − (LastByteSent − LastByteAcked).
  - That is, MaxWindow replaces AdvertisedWindow in the calculation of EffectiveWindow.
  - Thus, a **TCP source is allowed to send no faster than the slowest component—the network or the destination host—can accommodate.**

TCP Congestion Control

chi ce lo dice il congestion Windows?

- The problem, of course, is how TCP comes to learn an appropriate value for CongestionWindow.
- Unlike the AdvertisedWindow, which is sent by the receiving side of the connection, there is no one to send a suitable CongestionWindow to the sending side of TCP. → non posso chiederlo ai router intermedi perché non li conosco e non mi fido
- The answer is that the TCP source sets the CongestionWindow based on the level of congestion it **perceives** to exist in the network.
  ↳ deve osservare e calcolarsi la sua congestion window
- This involves decreasing the congestion window when the level of congestion goes up and increasing the congestion window when the level of congestion goes down.
- Taken together, the mechanism is commonly called **additive increase/multiplicative decrease (AIMD)**

TCP prova ad inviare più dati verificando se si verificano perdite di pacchetti

TCP Congestion Control

- **Additive Increase Multiplicative Decrease**
  - Mathematical Formula: Let
    - $w(t)$ = congestion window during time slot t
    - $a > 0$ = additive increase parameter
    - $0 < b < 1$ = multiplicative decrease factor
  - Then, the window at time slot t+1 is defined by:

$$w(t+1) = \begin{cases} w(t) + a & \text{if congestion is not detected} \\ w(t) \times b & \text{if congestion is detected} \end{cases}$$

aggiungo un pò alla mia finestra se al passo precedente non ho trovato congestione

1 MSS (di solito 1460 Byte)

se ho rilevato congestione allora rallento un pò

  - In TCP, a=1 MSS and b=0.5

TCP Congestion Control

- Additive Increase Multiplicative Decrease
  - How does the source determine that the network is congested and that it should decrease the congestion window?
  - Answer: observe that the main reason packets are not delivered (and a timeout results) is that *a packet was dropped due to congestion*.
    - It is rare that a packet is dropped because of an error during transmission.
    - (Observation not really true anymore with wireless connections...)
  - Therefore, TCP interprets **timeouts as a sign of congestion** and reduces the rate at which it is transmitting.
  - Specifically, each time a timeout occurs, the source sets CongestionWindow to half of its previous value (b=0.5)

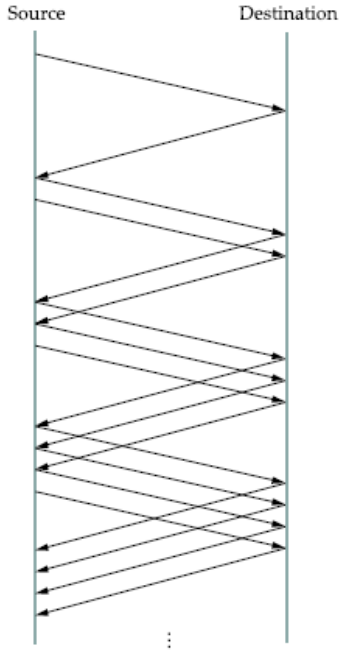
vedo se perdo pacchetti grazie al TIMEOUT

TCP Congestion Control

- Additive Increase Multiplicative Decrease
  - Although CongestionWindow is defined in terms of bytes, it is easiest to understand multiplicative decrease if we think in terms of whole packets.
  - For example, suppose the CongestionWindow is currently set to 16 packets. If a loss is detected, CongestionWindow is set to 8.
  - Additional losses cause CongestionWindow to be reduced to 4, then 2, and finally to 1 packet.
  - CongestionWindow is not allowed to fall below the size of a single packet, or in TCP terminology, the maximum segment size (MSS).

TCP Congestion Control

- Additive Increase Multiplicative Decrease
  - We also need to be able to increase the congestion window to take advantage of newly available capacity in the network.
  - This is the "additive increase" part of AIMD: Every time the source successfully sends a CongestionWindow's worth of packets—that is, each packet sent out during the last RTT has been ACKed—it adds the equivalent of 1 packet to CongestionWindow.



Packets in transit during additive increase, with one packet being added each RTT.

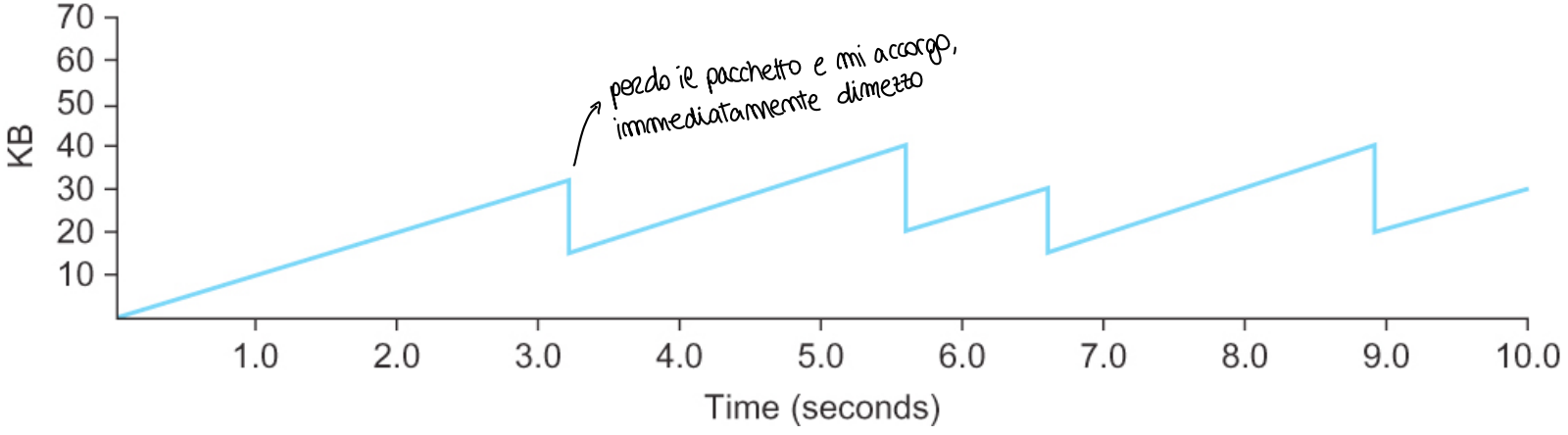
TCP Congestion Control

- Additive Increase Multiplicative Decrease
  - In practice, TCP does not wait for an entire window's worth of ACKs to add 1 packet's worth to the congestion window; instead increments CongestionWindow by a fraction of MSS every time an ACK is received.
  - Assuming that the received ACK acknowledges the receipt of (new) k bytes, then that fraction is k/CongestionWindow.
  - Specifically, each time an ACK of k bytes arrives the congestion window is incremented as follows:
    - Increment = MSS × (k/CongestionWindow)
    - CongestionWindow = CongestionWindow + Increment
  - In this way, since during a round (taking RTT) a whole CongestionWindow is sent, if all this data is ACKed the overall increment is exactly 1 MSS. If less data is ACKed, the overall increment is proportionally less than 1 MSS

in modo che se vengono inviati congestion Windows pacchetti (k = congwindow) senza errori, allora l'incremento è MSS, altrimenti se me perdo qualcuno k < congwindow e quindi l'incremento è < MSS

TCP Congestion Control

- Typical "sawtooth pattern" of CongestionWindow, in AIMD

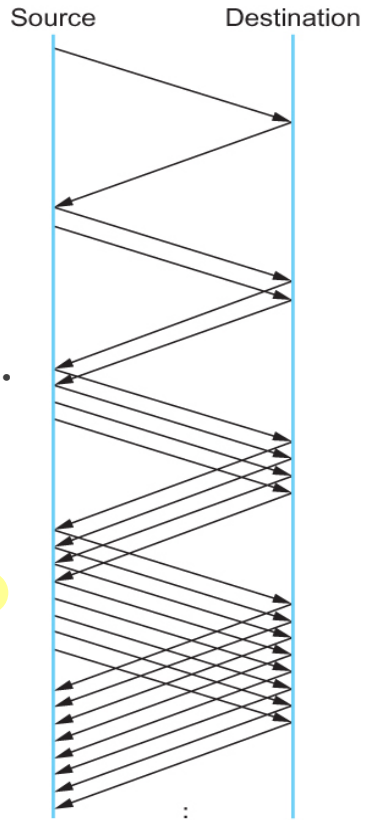


TCP Congestion Control

- Second ingredient: **Slow Start** → serve per partire più veloce, nome controintuitivo
  - The additive increase mechanism just described is the right approach to use when the source is operating close to the available capacity of the network, but it takes too long to ramp up a connection when it is starting from scratch.
  - TCP therefore provides a second mechanism, (ironically) called **slow start**, that is used to increase the congestion window rapidly from a cold start (e.g. right after handshake)
  - Slow start effectively increases the congestion window **exponentially**, rather than linearly.
    - Still slower than throwing an AdvertisedWindow amount of data at once, though!

TCP Congestion Control

- Slow Start raddoppio ad ogni ACK la congestion window e poi se perdo un pacchetto rallento
  - Specifically, the source starts out by setting CongestionWindow to one packet.
  - When the ACK for this packet arrives, TCP adds 1 to CongestionWindow and then sends two packets.
  - Upon receiving the corresponding two ACKs, TCP increments CongestionWindow by 2—one for each ACK—and next sends four packets.
  - The end result is that TCP effectively doubles the number of packets it has in transit every RTT.



TCP Congestion Control

- Slow Start
  - There are actually two different situations in which slow start runs.
  - The first is at the very beginning of a connection, at which time the source has no idea how many packets it is going to be able to have in transit at a given time.
    - In this situation, slow start continues to double CongestionWindow each RTT until there is a loss, at which time a timeout causes multiplicative decrease to divide CongestionWindow by 2.

TCP Congestion Control

- Slow Start
  - The second situation in which slow start is used is a bit more subtle; it occurs when the connection goes dead while waiting for a timeout to occur.
  - Recall TCP's sliding window algorithm: when a packet is lost, the source eventually reaches a point where it has sent as much data as the advertised window allows, and so it **blocks** while waiting for an ACK that will not arrive.
  - Eventually, a timeout happens, but by this time there are no packets in transit, meaning that the source will receive no ACKs to "clock" the transmission of new packets.
  - After to the timeout, the source will resend the lost packet, and receive a single cumulative ACK that reopens the entire advertised window
  - Now, dumping a whole window's worth of data on the network all at once would be too aggressive. Instead, the source uses slow start to restart the flow of data
  - (continues on next slide)

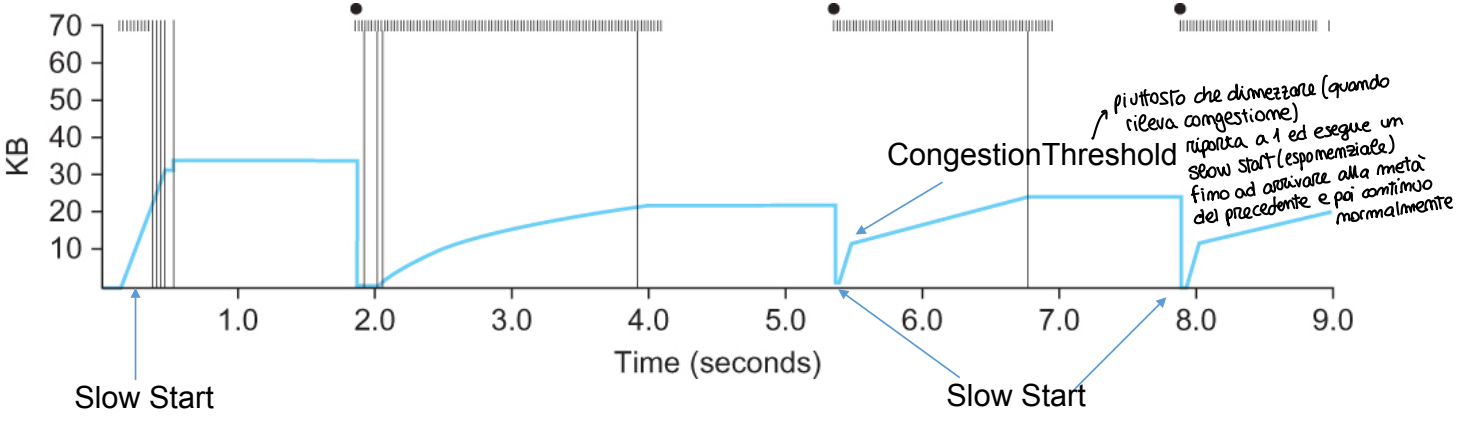
TCP Congestion Control

- Slow Start
  - After timeout, although the source is using slow start again, it now knows more information than it did at the beginning of a connection.
  - The source has a current (and useful) value of CongestionWindow; this is the value of CongestionWindow that existed prior to the last packet loss, divided by 2 as a result of the loss.
  - We can think of this as the “target” congestion window.
  - Slow start is used to rapidly increase the sending rate *up to this value*, and then additive increase is used beyond this point.

TCP Congestion Control

- Slow Start
  - We need to remember the "target" congestion window resulting from multiplicative decrease as well as the "actual" congestion window being used by slow start.
  - TCP introduces a temporary variable to store the target window, typically called *CongestionThreshold* or *ssthreshold*, that is set equal to the CongestionWindow value that results from multiplicative decrease.
  - The variable CongestionWindow is then reset to one packet, and it is incremented by one packet for every ACK that is received until it reaches CongestionThreshold, at which point it is incremented by one packet per RTT (as per normal AIMD).

TCP Congestion Control



Behavior of TCP congestion control in AIMD with slow start and no other technique.

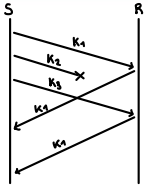
Colored line = value of CongestionWindow over time;
solid bullets at top of graph = timeouts;
hash marks at top of graph = time when each packet is transmitted;
vertical bars = time when a packet that was eventually retransmitted was first transmitted.

i BUCHI dove non trasmette sono fastidiosi perché sto sprecando banda

TCP Congestion Control

- Third ingredient: **Fast Retransmit**
  - The mechanisms described so far were part of the original proposal to add congestion control to TCP
  - It was soon discovered that the coarse-grained implementation of TCP timeouts led to long periods of time during which the connection went dead while waiting for a timer to expire.
  - Because of this, a new mechanism called **fast retransmit** was added to TCP (4.3BSD Unix "Tahoe", 1988).
  - Fast retransmit is a heuristic that sometimes triggers the retransmission of a dropped packet sooner than the regular timeout mechanism.

TCP Congestion Control



con il FAST TRANSMIT S rimvia K2 quando riceve 3ACK da R, quindi prima ancora che scada il timeout

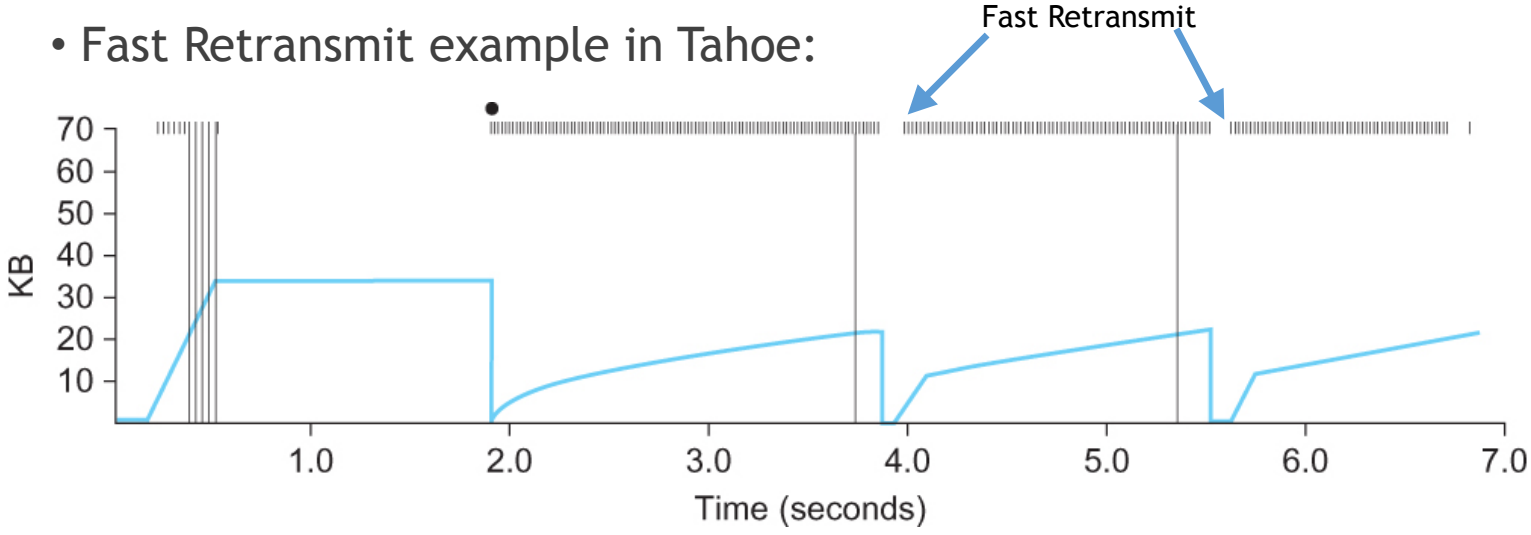
- Fast Retransmit
  - Recall that every time a data packet arrives at the receiving side, the receiver responds with an acknowledgment, even if this sequence number has already been acknowledged.
  - Thus, when a packet arrives out of order— that is, TCP cannot yet acknowledge the data the packet contains because earlier data has not yet arrived—TCP resends the same acknowledgment it sent the last time.
  - This second transmission of the same acknowledgment is called a **duplicate ACK**.

TCP Congestion Control

- Fast Retransmit
  - When the sending side sees a duplicate ACK, it knows that the other side must have received a packet out of order, which suggests that an earlier packet might have been lost.
  - Since it is also possible that the earlier packet has only been delayed rather than lost, the sender waits until it sees some number of duplicate ACKs and then retransmits the missing packet.
  - In practice (Tahoe and Reno), TCP waits until it has seen three duplicate ACKs before retransmitting the packet.

TCP Congestion Control

- Fast Retransmit example in Tahoe:

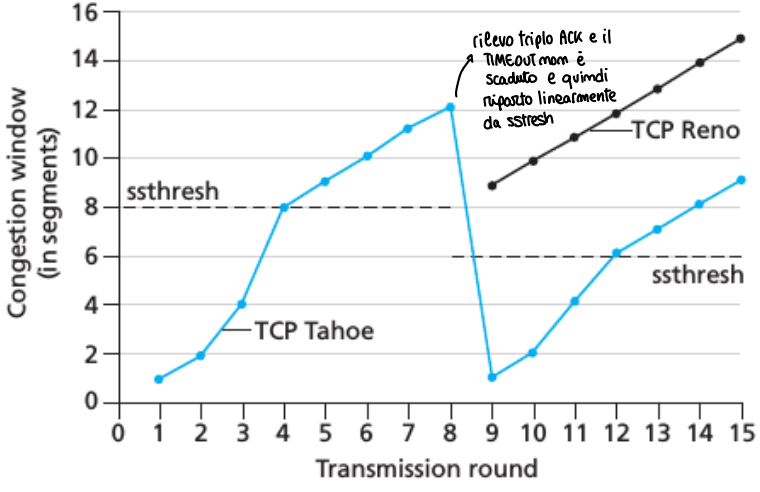


Trace of TCP with fast retransmit.
Colored line = CongestionWindow;
solid bullet = timeout;
hash marks = time when each packet is transmitted;
vertical bars = time when a packet that was eventually retransmitted was first transmitted.

TCP Congestion Control

risolve il fatto che bisogna ripartire da valori bassi ogni volta che ci sono ACK ripetuti, quando scade il timeout io torno sempre a uno in ogni caso e uso slow start
↑ prima che scada il timeout

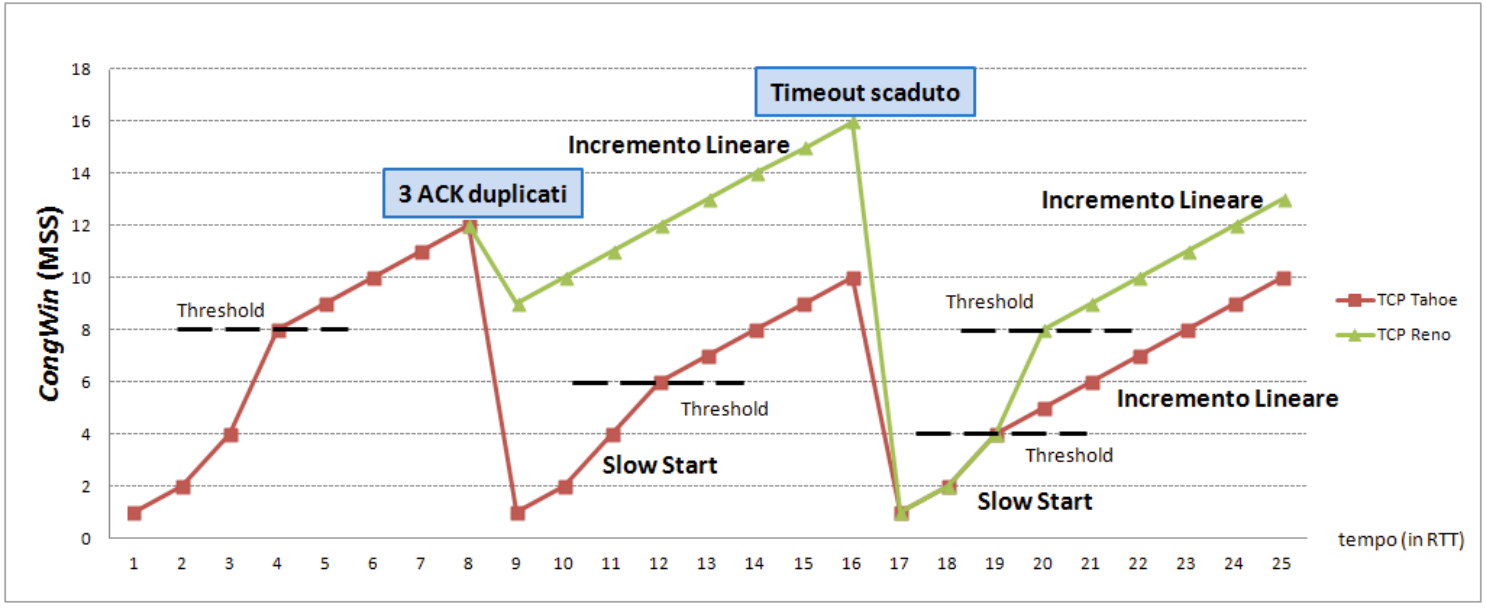
- Fourth ingredient: **Fast Recovery**
  - When the fast retransmit mechanism signals congestion, rather than drop the congestion window all the way back to one packet and run slow start, it is possible to use the ACKs that are still in the pipe to clock the sending of packets. → riparto dall'ssthreshold
  - This mechanism, called **fast recovery**, effectively removes the slow start phase that happens between when fast retransmit detects a lost packet and additive increase begins.
  - Implemented e.g. in Reno



TCP Congestion Control

- Fourth ingredient: **Fast Recovery**
  - When the fast retransmit mechanism signals congestion, rather than drop the congestion window all the way back to one packet and run slow start, it is possible to use the ACKs that are still in the pipe to clock the sending of packets.
  - This mechanism, called **fast recovery**, effectively removes the slow start phase that happens between when fast retransmit detects a lost packet and additive increase begins.
  - Implemented e.g. in Reno: if three duplicate ACKs are received:
    - execute the fast retransmit
    - set the ssthresh to the half of the congestion window
    - set the new congestion window to sshthresh + 3 MSS, instead of setting it to 1 MSS like Tahoe *(fast recovery)*
  - In any case, if an ACK times out, reduce congestion window to 1 MSS and resets to slow start state

TCP Congestion Control



TCP Congestion Control: exercises

non c'è fast recovery.

Un certo host TCP, che usa un algoritmo di congestion control con partenza lenta e ritrasmissione veloce, apre una nuova connessione, con MSS=1400. Esegue 3 round completi in partenza lenta, partendo da `CongestionWindow` = 2 MSS=2800. Se non vengono ricevuti gli ACK per due segmenti inviati nel terzo round, quanto vale `CongestionWindow` all'inizio del quarto round?

R: Partendo con `CongestionWindow`=2 MSS, dopo il primo round `CongestionWindow`=4 MSS; dopo il secondo round `CongestionWindow`=8 MSS; dopo il terzo round `CongestionWindow`=16 MSS meno 2 = 14 MSS = 19600.

non slow start, aggiungo solo uno

Una connessione TCP, con MSS=1400 e recupero veloce, si trova nella fase additiva, e inizia il round con CongestionWindow = 10000. Dopo aver inviato un po' di segmenti lunghi MSS e ricevuto 5 ACK, scade il timeout di un segmento inviato in precedenza. Quanto diventa CongestionWindow?

R: L'incremento ricevuto per ogni ACK è di MSS*MSS/CW = 196 byte. Quindi, dopo 5 segmenti riscontrati, CW è diventato 10000+196*5 = 10980. A questo punto scade il timeout, e quindi CW viene dimezzata diventando 5490.

Congestion Avoidance Mechanism

Finora lasciavamo andare in congestione e poi risolvevamo. Ora cerchiamo di evitare che avvenga la congestione

- It is important to understand that TCP's strategy is to **control congestion** once it happens, as opposed to trying to avoid congestion in the first place.
- In fact, TCP repeatedly increases the load it imposes on the network in an effort to find the point at which congestion occurs, and then it backs off from this point.
- An appealing alternative, but not widely adopted yet, is **to predict when congestion is about to happen and then to reduce the rate at which hosts send data just before packets start being discarded**.
- We call such a strategy **congestion avoidance**, to distinguish it from congestion control.

Congestion Avoidance Mechanism

- Random Early Detection (RED)
  - Invented by Sally Floyd and Van Jacobson in the early 1990s
  - A mechanism called where each router is programmed to monitor its own queue length, and when it detects that congestion is imminent, to notify the source to adjust its congestion window.
  - Rather than explicitly sending a congestion notification message to the source, RED is most commonly implemented such that it **implicitly** notifies the source of congestion by dropping one of its packets. → butta pacchetti senza che sia in congestione. Facendo così raccomta
  - The source is, therefore, effectively notified by the subsequent timeout or duplicate ACK.
  - RED is designed to be used in conjunction with TCP, which currently detects congestion by means of timeouts (or some other means of detecting packet loss such as duplicate ACKs). ⟶ viene usato anche con UDP

Congestion Avoidance Mechanism

- Random Early Detection (RED)
  - As the "early" part of the RED acronym suggests, the gateway drops the packet earlier than it would have to, so as to notify the source that it should decrease its congestion window sooner than it would normally have.
  - In other words, **the router drops a few packets before it has exhausted its buffer space completely**, so as to cause the source to slow down, with the hope that this will mean it does not have to drop lots of packets later on.

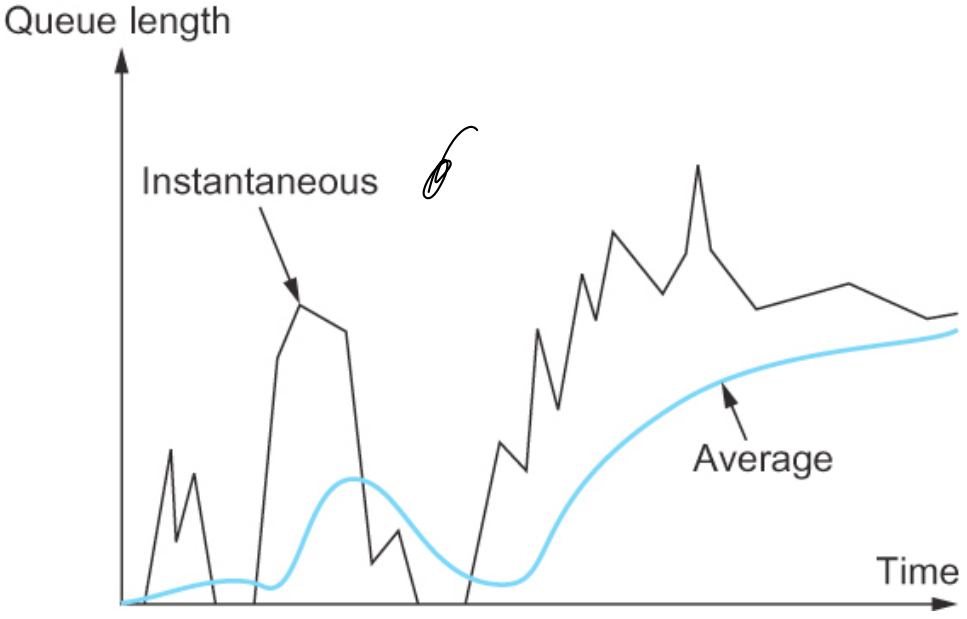
Congestion Avoidance Mechanism

- Random Early Detection (RED)
  - how RED decides when to drop a packet and what packet it decides to drop? ⟶ ogni volta che arriva un pacchetto c'è una probabilità (in base all'occupaz. della coda, se è quasi piena la prob. è più alta) che lo scarti
  - To understand the basic idea, consider a simple FIFO queue. Rather than wait for the queue to become completely full and then be forced to drop each arriving packet, we could decide to drop each arriving packet with some drop probability whenever the queue length exceeds some drop level.
  - This idea is called **early random drop**. The RED algorithm defines the details of how to monitor the queue length and when to drop a packet.

Congestion Avoidance Mechanism

- Random Early Detection (RED)
  - First, RED computes an average queue length using a weighted running average similar to the one used in the original TCP timeout computation. That is, AvgLen is computed as
    - AvgLen = (1 − w) × AvgLen + w × SampleLen
    - where 0 < w < 1 and SampleLen is the length of the queue when a sample measurement is made.
  - In most software implementations, the queue length SampleLen is measured every time a new packet arrives at the gateway.
  - In hardware, it might be calculated at some fixed sampling interval.

uso la media MOBILE al posto che quella istantanea



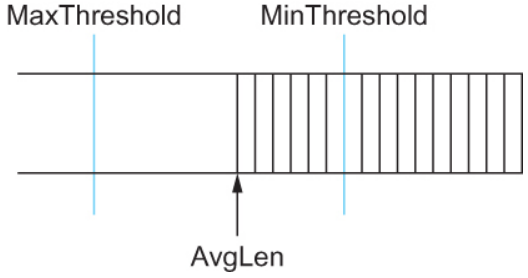
Congestion Avoidance Mechanism

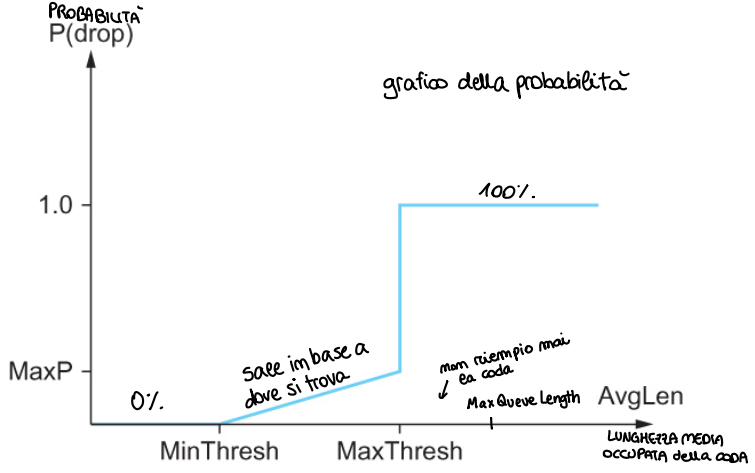
- Random Early Detection (RED)
  - Second, RED has two queue length thresholds that trigger certain activity: **MinThreshold** and **MaxThreshold**.
  - When a packet arrives at the gateway, RED compares the current AvgLen with these two thresholds, according to the following rules:
    - if AvgLen ≤ MinThreshold ⟶ sono safe
      - queue the packet
    - if MinThreshold < AvgLen < MaxThreshold ⟶ calcolo la prob. di scartarlo
      - calculate probability P(drop)
      - drop the arriving packet with probability P(drop)
    - if MaxThreshold ≤ AvgLen ⟶ sono già oltre al limite, scarta
      - drop the arriving packet

Facendo così RED riduce il DELAY (↗ PING) perché mantiene le code più corte

Congestion Avoidance Mechanism

- Random Early Detection (RED)
  - P(drop) is computed as follows (where MaxP is a parameter):
    P(drop) = MaxP × (AvgLen − MinThresh)/(MaxThresh − MinThresh)





Drop probability function for RED

RED: examples

Se non lo dice MAXP=1

Un utente si accorge che un certo router, che implementa la RED con MinThreshold=20KB e MaxThreshold=100KB, perde circa il 10% dei pacchetti. Quanto è piena la coda sul router, in kB?

R: Per fare una probabilità di 0.1, bisogna che sia $0.1 = \frac{x-20}{100-20}$ (AVGLEN), quindi $8 = x - 20$ da cui $x = 28$ KB.

siamo molto lontani dai 100KB di coda piena eppure ha probabilità del 10% di essere scartati

Un router applica la politica RED ad una coda, con Minthreshold=10kB e Maxthreshold=20kB. La coda attualmente è piena a 9kB. Arrivano in rapida successione tre pacchetti, di rispettivamente 2kB, 3kB e 2kB. Qual è la probabilità che tutti e tre vengano accodati? (non contiamo i pesi average dei pacchetti, li sommo cumulativamente)

R: Il primo pacchetto viene accodato sicuramente (perché AvgLen < MinThresh, ora però AvgLen=11) quindi $P_1 = 1$, e la coda si allunga a 11kB. La probabilità che il secondo pacchetto sia scartato è $(11-10)/(20-10)=1/10$, quindi $P_2 = 1-1/10=0,9$ (1 - prob. che sia scartato). Se il secondo pacchetto è stato accodato, la coda diventa 14kB. La probabilità che il terzo pacchetto sia scartato è $(14 - 10)/(20 - 10) = 4/10$, quindi $P_3 = 1 - 4/10 = 0,6$. La probabilità che tutti e tre i pacchetti vengano accodati è quindi $P = P_1P_2P_3 = 1*0,9*0,6=0,54=54\%$.

Congestion Avoidance Mechanism

- **Source-based Congestion Avoidance**

tecnica RED che è implementata nei ROUTER
ma si possono anche implementare negli host

  - The general idea of these techniques is to watch for some sign from the network that some router's queue is building up and that congestion will happen soon if nothing is done about it.
  - For example, the source might notice that as packet queues build up in the network's routers, there is a measurable increase in the RTT for each successive packet it sends.

Congestion Avoidance Mechanism

- Source-based Congestion Avoidance
  - One particular algorithm exploits this observation as follows:
    - The congestion window normally increases as in TCP, but every two round-trip delays the algorithm checks to see if the current RTT is greater than the average of the minimum and maximum RTTs seen so far.
    - If it is, then the algorithm decreases the congestion window by one-eighth.

      CurrentWindow = CurrentWindow * 0,875

Congestion Avoidance Mechanism

- Source-based Congestion Avoidance
  - A second algorithm: the decision as to whether or not to adjust the current window size is based on *changes* to both the RTT and the window size.
  - The window is adjusted once every two round-trip delays based on the product
    $$(\text{CurrentWindow} - \text{OldWindow}) \times (\text{CurrentRTT} - \text{OldRTT})$$
  - If the result is positive, it means that either the window has enlarged BUT the RTT has increased, or that the RTT has decreased BUT the window has reduced
    - then, the source decreases the window size by one-eighth;
  - if the result is 0 or negative, the network is stable or even faster than before
    - then, the source increases the window by one maximum packet size.
  - Note that the window changes during every adjustment; that is, it oscillates around its optimal point.
  - Can avoid to use packet losses to recognise congestions
    - Used (with variants) in Vegas, BBR, …

Summary

- We have discussed different resource allocation mechanisms
- We have discussed different queuing mechanisms
- We have discussed TCP Congestion Control mechanisms
- We have discussed Congestion Avoidance mechanisms
- ~~We have discussed Quality of Services~~