

Tema 1 - El protocol TCP a fons

Sergi Robles

Sergi.Robles@uab.cat

Departament d'Enginyeria de la Informació i de les Comunicacions
Universitat Autònoma de Barcelona

Tecnologies avançades d'Internet

Contingut

- 1 Repàs TCP
- 2 Timeouts i retransmissions
- 3 Resposta a la congestió
- 4 Finestres menguants
- 5 Millores de TCP i rendiment

Contingut

- 1 Repàs TCP
 - Xarxes de Computadors II
- 2 Timeouts i retransmissions
- 3 Resposta a la congestió
- 4 Finestres menguants
- 5 Millores de TCP i rendiment



Vegeu transparències de Xarxes de
Computadors II, Tema 3, secció 3.

Contingut

1 Repàs TCP

2 Timeouts i retransmissions

- Introducció
- Timeout adaptatiu
- Keepalive

3 Resposta a la congestió

4 Finestres menguants

5 Millores de TCP i rendiment

Les idees més importants i complexes de TCP estan en el càlcul dels *timeouts* i les retransmissions.



Quan TCP envia un segment posa en marxa un *timer* associat.

- Si el *timer* acaba abans d'arribar la confirmació s'assumeix que el segment s'ha perdut i el retransmet.
- Si la confirmació arriba abans d'expirar el *timer* s'envia el següent segment.

Molt protocols utilitzen uns temps predefinits i constants.
Per què TCP ha de ser diferent dels altres?

Molt protocols utilitzen uns temps predefinits i constants.
Per què TCP ha de ser diferent dels altres?

→ Perquè està dissenyat per a una internet!

- No sabem quins poden ser els endarreriments
- La freqüència d'arribada de confirmacions no és constant
- Depèn del tràfic: d'un moment a un altre pot ser totalment diferent

Contingut

- 1 Repàs TCP
- 2 Timeouts i retransmissions
 - Introducció
 - Timeout adaptatiu
 - Keepalive
- 3 Resposta a la congestió
- 4 Finestres menguants
- 5 Millores de TCP i rendiment



El protocol TCP utilitza un algorisme de retransmissió adaptatiu.

El rendiment de la connexió és monitoritzat per tal d'ajustar el millor valor possible pel *timeout*. S'adapta als canvis.

- S'agafa el valor entre el segment enviat i la seva confirmació (*roundtrip time (RTT)* de mostra).
- Es va fent una mitja ponderada per a que el temps estimat d'un *roundtrip time* vagi variant.

Una primera tècnica va ser triar un α ($0 \leq \alpha < 1$) per ponderar l'antic RTT estimat respecte el RTT de mostra més nou a l'hora de calcular la nova estimació.

Primera aproximació

$$SRTT = (\alpha * SRTT) + ((1 - \alpha) * RTT_m)$$

on $SRTT$ és el *Smooth RTT* i RTT_m és el *RTT* mesurat

- Si α és a prop de 1 → El RTT estimat no varia per canvis que duren poc.
- Si α és a prop de 0 → S'ajusta ràpidament als canvis.

De manera totalment equivalent, podem escriure també:

$$SRTT = \alpha * SRTT + (1 - \alpha) * RTT_m$$

$$SRTT = SRTT + (\alpha - 1) * SRTT + (1 - \alpha) * RTT_m$$

$$SRTT = SRTT - (1 - \alpha) * SRTT + (1 - \alpha) * RTT_m$$

i si substituem $\delta = (1 - \alpha)$ tenim que

$$SRTT = SRTT + \delta * (RTT_m - SRTT)$$

No hi ha cap diferència entre les dues maneres d'escriure l'expressió.

Aquesta mitja ponderada es diu **EWMA** (*Exponentially Weighted Moving Average*), i s'utilitza també en altres contextos.

Per a calcular el temps d'espera (*timeout*) es solia utilitzar un factor β constant ($\beta > 1$).

$$\text{Timeout} = \beta * RTT$$

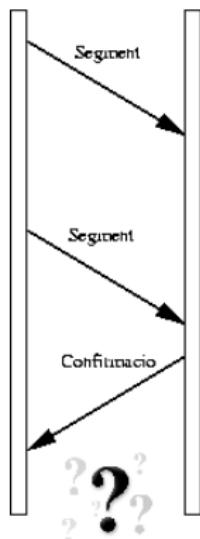
- Un bon factor β era molt difícil de trobar!
- Per a detectar pèrdues ràpidament necessitariem un β proper a 1, però aleshores petits endarreriments provocarien la retransmissió i s'utilitzaria molta amplada de banda.
- A l'especificació original de TCP → $\beta \in [1.3, 2]$. Ara hi ha tècniques millors per ajustar el valor del *timeout*.

→ Com calculem exactament les mostres de RTT?

→ El problema no és tan senzill com sembla. Recordem que TCP utilitza un esquema de confirmació acumulatiu de dades del flux, no de datagrames específics!

Si el temporitzador expira després d'enviar un segment es torna a enviar. Quan arriba la confirmació **no sabem** si és del primer segment enviat o del segon! (ambigüïtat de confirmació).

- Què fem, assumim que és una confirmació del primer segment, o del segon segment?





→ No funcionaria si assumissim qualsevol dels dos casos!!

Cas 1

- Imaginem que associem la confirmació a la transmissió original. Si hi ha **pèrdues** de datagrames el RTT aniria fent-se **gran** i més gran!

El nou RTT serà calculat utilitzant un RTT massa gran, així que es farà més gran poc a poc. El *timeout* també serà més gran, i per tant el següent RTT de mostra també. La estimació de RTT anirà creixent i creixent.

Cas 2

- Si associessim la confirmació a la retransmissió més recent tampoc funcionaria.

Si en un moment donat hi ha un **endarreriment** d'extrem a extrem (els datagrames triguen més en arribar però no es perden) el *timeout* **expirarà** abans que arribi la confirmació. Es tornarà a **re-enviar** el segment i arribarà la confirmació del primer segment.

El *roundtrip* de mostra serà molt petit, baixant l'estimació del RTT. El següent *timeout* serà més petit també. El RTT correcte sol ser una mica més gran que algun múltiple del valor RTT estimat, un cop s'estabilitza.

Si no serveix associar la confirmació ni al segment original i al retransmès... Què fem? Hi ha alguna solució?

Si no serveix associar la confirmació ni al segment original i al retransmès... Què fem? Hi ha alguna solució?

→ **Algorisme de Karn!**

Algorisme de Karn

No actualitzem l'estimació del RTT pels segments retransmesos. Utilitzem únicament les confirmacions no ambigües.



Si ho fem de manera massa simplista també fallarà: Si l'endarreriment augmenta es re-enviaran segments i l'estimació mai serà més gran.

Per arreglar això, l'algorisme de Karn combina una estratègia de backoff del timeout:

Per a cada retransmissió, TCP incrementa el *timeout* (amb el límit superior de l'endarreriment màxim (120 segons)). Quan arriba la primera confirmació a un segment no retransmès, el *timeout* torna al seu valor anterior (*backoff*).

A la majoria d'implementacions tenim:

$$\text{newTimeout} = \gamma * \text{Timeout}, \text{ on normalment } \gamma = 2$$

L'algorisme de Karn combina:

- Estimació de RTT només de no retransmesos
- *Backoff* del *timeout* quadràtic.

Aquest algorisme és bó fins i tot quan hi ha moltes pèrdues.

El mecanisme utilitzat a la pràctica varia d'una implementació a una altra (netBSD va multiplicant per 2 fins a 64, després suma, Linux va multiplicant per 2, fins al màxim de 120, ...).

Exemple: Linux

```
tp->rto = min(tp->rto << 1, TCP_RTO_MAX);  
tcp_reset_xmit_timer(sk, TCP_TIME_RETRANS, tp->rto);
```

Fins ara, els mecanismes vistos no s'adapten si hi ha molta variació en l'endarreriment. → S'assumeix que la variança és petita i constant.

A la pràctica

Els RTT creixen amb la càrrega de la xarxa, i el pitjor, la variança σ del RTT creix molt més!

$$\sigma \propto \frac{1}{(1-L)}, \text{ on } L \text{ és la càrrega de la xarxa, } 0 \leq L \leq 1$$

Amb l'adaptació original $RTO = \beta * RTT$, amb $\beta = 2$ normalment, no és suficient.

A partir d'una càrrega de xarxa del **30%** deixa de funcionar de manera correcta (!): es pensa que les confirmacions arribaran i retransmet el segment després del *timeout*, **incrementant** encara més la càrrega!!



La nova especificació de TCP té en compte **l'estimació de la variança** i del RTT. L'estimació de la variança farà el paper de β .

→ La capacitat de TCP (*throughput*) s'incrementa molt.



Algorisme de Van Jacobson

S'aprofita l'estimació de la variança del RTT per a calcular un temps de *timeout* que s'adapti a condicions de càrrega de xarxa alta.



$mdev_m = |RTT_m - RTT|$ mean deviation mesurada

$SRTT = SRTT + \delta * (RTT_m - SRTT)$

$mdev = mdev + \rho * (mdev_m - mdev)$)

$RTO = SRTT + \eta * mdev$

SRTT: és el RTT de variació suau (*smooth*).

δ : Cóm afecta la nova mostra al RTT, $0 \leq \delta \leq 1$

ρ : Velocitat d'adaptació de la variació, $0 \leq \rho \leq 1$

η : Cóm la variació afecta al *timeout*.

- A la variança $mdev$ també se li sol dir *jitter*.
- Per a fer els càlculs més ràpids, δ i ρ soLEN ser inversos de potències de 2 ($\frac{1}{2^n}$).
- Els valors empírics suggerits per a l'algorisme són: $\delta = \frac{1}{2^3}$, $\rho = \frac{1}{2^2}$, i $\eta = 2, 3$, ó 4.

→ Si hi ha **pocs canvis** entre el RTT de mostra i el RTT estimat ($SRTT$ és molt similar a RTT_m durant molt temps), **$mdev$** estarà **proper a 0** i el SRTT serà una bona estimació pel RTO (si triga més de SRTT, hi ha retransmissió).

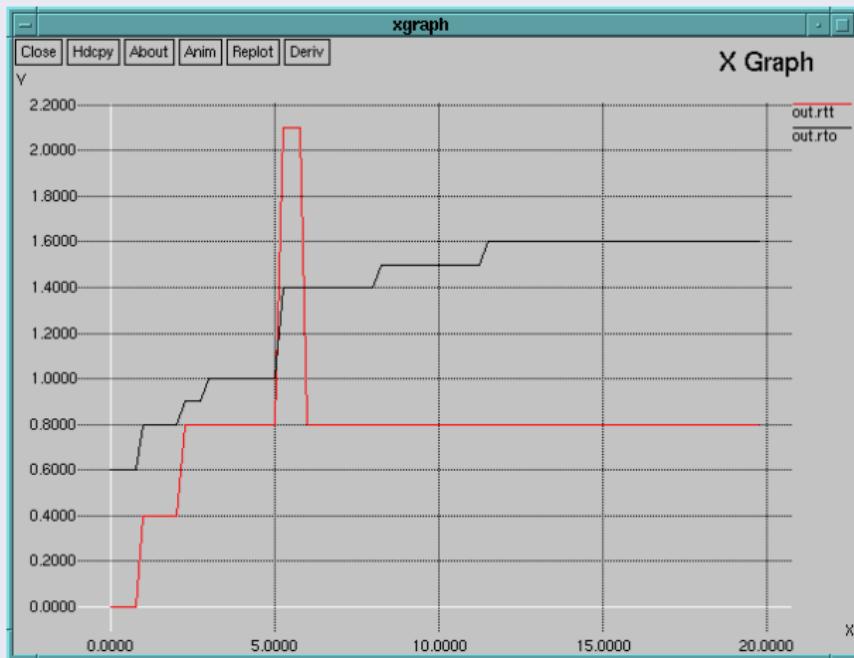
Si hi ha molts canvis, no ens fiem, i tenim un marge gran, proporcional a la nostra incertesa.

Veiem dues gràfiques: amb el mecanisme original, i amb Jacobson.

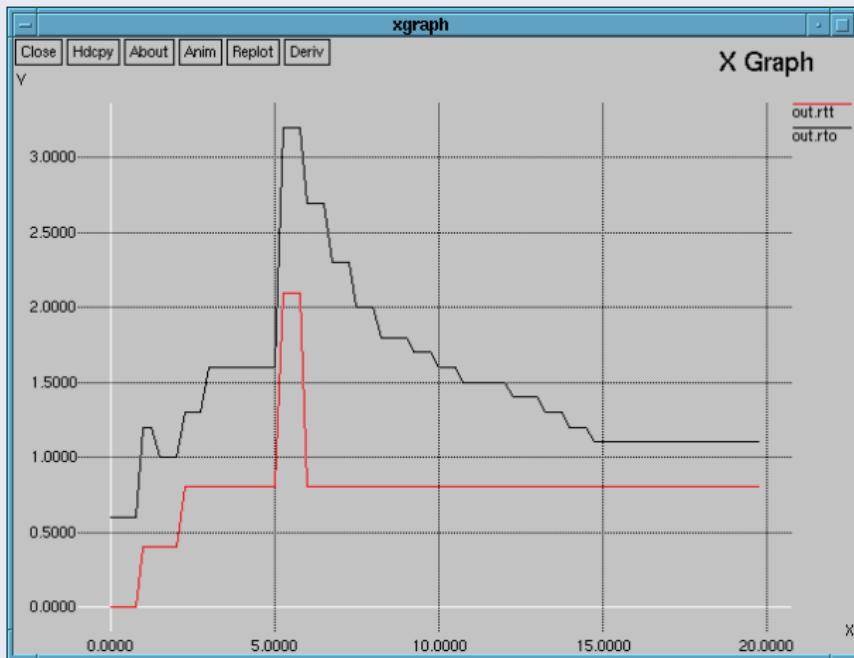
- A l'original el RTO **no reacciona prou ràpid** i no li dóna temps de pujar. Després li costa baixar.
- Amb l'algorisme de Jacobson el canvi és detectat ràpidament i el **RTO s'incrementa a temps**. Després, torna a baixar.

→ Si $\text{RTO} < \text{RTT}$ es produeix una retransmissió innecessària!

Algorisme original:



Algorisme de Jacobson:



Contingut

- 1 Repàs TCP
- 2 Timeouts i retransmissions
 - Introducció
 - Timeout adaptatiu
 - Keepalive
- 3 Resposta a la congestió
- 4 Finestres menguants
- 5 Millores de TCP i rendiment

Pot resultar sorprenent que durant una connexió TCP en la que no s'estan intercanviant dades (*idle*) no circula **ni un sol segment.**

Una connexió pot romandre oberta durant mesos sense intercanviar ni un sol bit. En aquest temps poden haver-se canviat routers, penjat i restaurat línies telefòniques, etc.

Keepalive

→ Tot i que la detecció de l'estat de l'altre extrem correspon a les aplicacions, algunes implementacions de TCP incorporen un temporitzador de *keepalive*.

Els *keepalive* no són part de l'especificació de TCP:

- Poden tancar bones connexions degut a fallades temporals.
- Consumen amplada de banda innecessaria.
- Costen diners si el proveïdor d'Internet ens cobra per datagrama.

D'altra banda, resulten força convenients quan els clients no estan sempre engegats: PCs, portàtils, PDAs, poden “desapareixer” deixant connexions obertes. Sense els *keepalive* el servidor mantindria aquestes connexions per sempre.

Funcionament:

Si no hi ha activitat després de dues hores de connexió, el servidor envia un segment de **sondeig**. Poden passar 4 coses:

- **El client està corrent de manera normal.** Respondrà i el servidor tornarà a sondejar després de dues hores.
- **El client està aturat.** El servidor envia fins a 10 sondejos cada 75 segons abans d'acabar la connexió.
- **El client ha rebootat.** Enviarà un reset al servidor.
- **El client està bé, però no li arriben els segments.** És equivalent al segon cas.



Contingut

- 1 Repàs TCP
- 2 Timeouts i retransmissions
- 3 Resposta a la congestió
 - Introducció
 - Cues
 - Tail Drop
 - RED
 - Altres cues
- 4 Finestres menguants
- 5 Millores de TCP i rendiment

Resposta a la congestió

TCP no només considera endarreriments entre els punts finals de la connexió: també reacciona a la congestió de la internet.

Congestió:

Condició d'**endarreriments** importants causada per una sobrecàrrega de datagrames en un o més elements de commutació (p.e. *routers*).



Si hi ha congestió, l'endarreriment augmenta i els datagrames fan cua al router (mentre aquest els pugui mantenir (tenen memòria limitada!)).

Des de TCP només és possible apreciar els endarreriments.
S'ignoren totalment les causes.

Si quan hi ha congestió tothom re-enviés els segments dels que no s'ha rebut confirmació, es produiria encara més congestió, fins al col·lapse!

Mecanismes anti-congestió

→ TCP pot ajudar a disminuir la congestió aplicant **mecanismes i estratègies** d'enviament de dades que disminueixin la probabilitat de sobrecàrrega de la xarxa i que permetin la seva recuperació.

TCP assumeix que l'expiració del *timeout* implica una pèrdua del segment deguda a congestió. S'ha d'anar amb molt de compte per a evitar falsos positius*, ja que a Internet hi ha variacions molt grans en els RTT.

L'estàndard de TCP recomana tres tècniques molt fàcils d'implementar:

- **Decrement multiplicatiu** (*Multiplicative decrease*)
- **Començament lent** (*Slow start*)
- **Fase d'evitar congestió**

* Es detecta una pèrdua per congestió quan en realitat només hi havia un increment del RTT degut a un altre motiu.

Si hi ha congestió no és bona idea enviar un flux de dades massa gran (la probabilitat de que no arribi serà proporcional a la mida).

Finestra de Congestió

→ A més de la finestra d'emissió, TCP ha de mantenir una **finestra de congestió** per a reduir el flux de dades per sota de la finestra lliscant quan hi ha congestió.

Inicialment, i si no hi ha congestió, el tamany d'aquestes dues finestres és el mateix. Només es podran enviar dades de la finestra permesa:

$$F. \text{ Permesa} = \min(F. \text{ Anunciada}, F. \text{ Congestió})$$

Recordem: TCP assumeix que la pèrdua de datagrames és causada per congestió.

Decrement multiplicatiu: Si hi ha pèrdua divideix per dos la finestra de congestió (com a mínim serà d'un segment). Pels que queden, augmenta el RTO exponencialment.

Objectiu: reduir el tràfic per a donar temps a que es redueixin les cues als routers.

→ Cal algun mecanisme per a recuperar-se després d'una congestió...

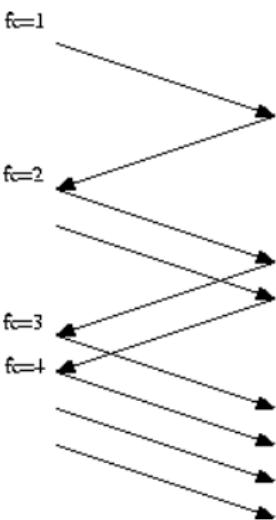
Començament lent: Per a començar el tràfic en una nova connexió, o per a incrementar el tràfic després d'una congestió, s'utilitza aquest mecanisme.

Es posa la finestra de congestió a un sol segment. S'incrementa en un segment cada vegada que arriba una confirmació.

→ **Objectiu:** evitar col.lapsar la xarxa tot just comencem a enviar un flux de dades, o després d'una congestió.

Deprés del SYN+ACK en l'establiment de connexió, ja podrem enviar 2 segments. Quan arribin les seves confirmacions, 4 segments més, i així successivament.

→ Normalment amb **4 RTT** (15 segments) és **suficient** per a arribar a la mida de la finestra anunciada.



L'estàndard (RFC 3390) especifica que la finestra de congestió inicial no hauria de superar:

$$\min(4 * MSS, \max(2 * MSS, 4380\text{bytes}))$$

Que en el cas d'Ethernet (MTU 1500, MSS=1460 si no hi ha opcions) són 3 MSS.

Actualment s'està discutint l'ampliació d'aquest valor a 10 MSS, donat que les velocitats de les xarxes ara són molt altes i hi ha molt tràfic de connexions curtes (Google és el principal impulsor).

→ A la pràctica trobem diferències entre les implementacions, essent 2, 3 i 4 MSS els valors més freqüents.

Fase d'evitar congestió: TCP entra en aquesta fase si la finestra de congestió (CWND) arriba a un cert nivell *ssthresh*.

El que es fa exactament en aquesta fase de congestió depèn de la implementació particular de TCP.

Alguns exemples d'accions en aquesta fase:

TCP Tahoe: Quan es reben 3 ACK duplicats, *Fast Retransmission* (S'envia el segment que sembla haver-se percut, sense esperar el timeout), es disminueix CWND 1 MSS, i es fa *Slow Start*.

TCP Reno: Quan 3 ACK duplicats, *Fast Retransmission*, CWND a la meitat, *Fast Recovery* (es re-envia el segment, i s'espera el ACK de tota la finestra de transmissió). Si TMOUT, CWND = 1MSS i *Slow Start*.

TCP New Reno: Igual que Reno, però en el *Fast Recovery* per cada ACK duplicat rebut s'envia un nou segment del final de la CWND, i per cada ACK de confirmació parcial s'envia en següent segment. Normalment es combina amb l'opció *Selective ACK* (SACK).

TCP CUBIC: Optimitza el control de la congestió per xarxes ràpides amb *delay* gran, fent que la finestra sigui una funció cúbica del temps des de la darrera congestió (Linux 2.6.19-).

Els mecanismes descrits: **Decrement multiplicatiu**, **Començament lent**, **Fase d'evitar congestió**, juntament amb el **Backoff del timeout** (Karn) i la **Mesura de la variació** (Jacobson), milloren el rendiment de TCP de manera molt significativa (de 2x fins a 10x!!) sense afegir massa computació addicional.

Encara es van afegint modificacions i altres algorismes per a evitar la congestió...



Contingut

- 1 Repàs TCP
- 2 Timeouts i retransmissions
- 3 Resposta a la congestió
 - Introducció
 - Cues
 - Tail Drop
 - RED
 - Altres cues
- 4 Finestres menguants
- 5 Millores de TCP i rendiment

Encara que les capes de protocols siguin independents les unes de les altres, els comportaments d'uns protocols **affecten** d'altres en molts casos.

Pel disseny de nous mecanismes en un protocol, tots els altres protocols de la resta de capes han de ser **reconsiderats** també!!



→ La interacció més important entre IP i TCP és deguda a la gestió de les cues d'IP.

Un *router* tria el camí que ha de seguir un datagrama segons els mecanismes que hem anat veient. Mentre un datagrama és processat, la resta de datagrames que arriben esperen fent "cua".

→ Donat que la memòria dels routers no és infinita, caldrà una política de gestió de cues específica per a determinar què es fa si arriben més datagrames dels que caben a la cua (quins es descarten).

Les polítiques més habituals són:

- Tail drop (**FIFO**)
- Random Early Discard (**RED**)

Contingut

- 1 Repàs TCP
- 2 Timeouts i retransmissions
- 3 Resposta a la congestió
 - Introducció
 - Cues
 - Tail Drop
 - RED
 - Altres cues
- 4 Finestres menguants
- 5 Millores de TCP i rendiment

Política Tail Drop:

Es descarten els datagrames que arriben si la cua ja està plena (s'elimina la cua).

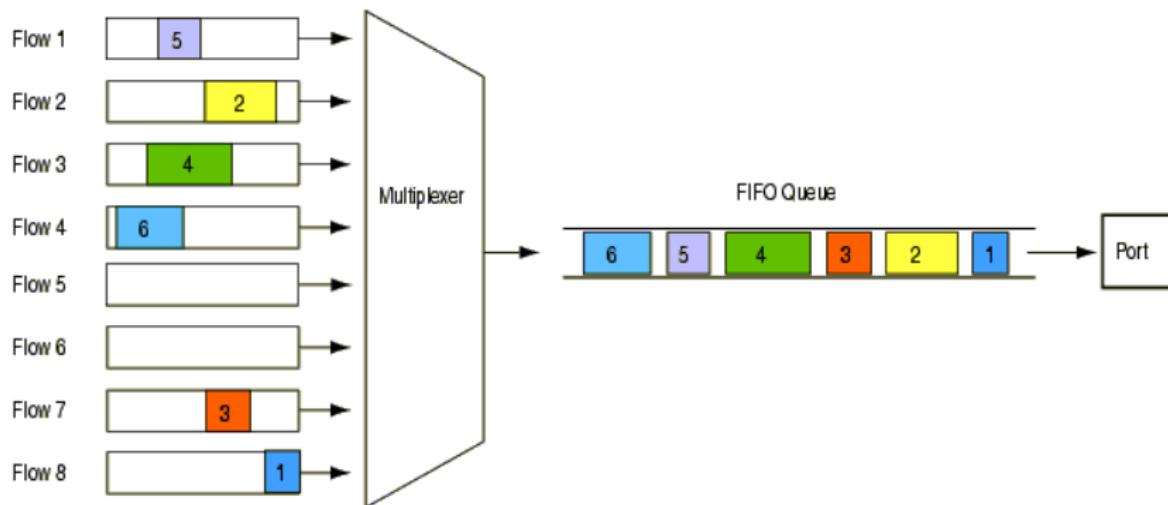
També s'anomena *pfifo* (per datagrames) o *bfifo* (per bytes).



→ Problema: **Sincronització global**. Es perden els datagrames de N connexions, i totes entren en començament lent al mateix temps.

L'únic paràmetre d'aquesta política és la mida de la cua.

Esquema del funcionament de la cua de Tail Drop (FIFO):



Contingut

- 1 Repàs TCP
- 2 Timeouts i retransmissions
- 3 Resposta a la congestió
 - Introducció
 - Cues
 - Tail Drop
 - RED
 - Altres cues
- 4 Finestres menguants
- 5 Millores de TCP i rendiment

Una de les alternatives per a evitar els problemes de Tail Drop és RED.

Política de Random Early Discard (RED):

Alguns datagrames **es descarten abans** d'haver arribat a la capacitat màxima de la cua. D'aquesta manera s'evita el problema de la sincronització global.

Existeixen dos límits en la cua, a part de la capacitat:

$$T_{min} \text{ i } T_{max}$$

El comportament de la cua dependrà de la mida actual i d'aquests límits.

Quan arriba un nou datagrama es determina la seva posició així:

- Si la cua té menys de T_{min} datagrames, afegir-lo.
- Si la cua té més de T_{max} , descartar-lo.
- Si en té entre T_{min} i T_{max} , descartar el datagrama segons una probabilitat p .

→ D'alguna manera és com si RED simulés una congestió abans de que es propueixi per a no haver de descartar de manera obligada els últims datagrames.

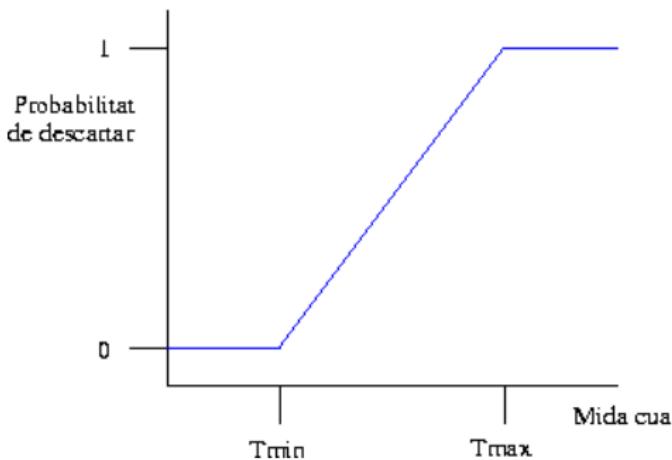
La clau per a que RED funcioni bé és triar uns bons valors per T_{min} , T_{max} , i p .

Aquestes són algunes consideracions per a assignar paràmetres a una cua RED:

- T_{min}** : Hauria de ser **gran**, per a que hi hagi una bona utilització de l'enllaç.
- T_{max}** : Més gran que T_{min} en més de l'increment habitual durant un RTT (2 vegades T_{min}).
- p** : Calcular aquesta probabilitat és el més **complex** d'aquesta cua.

En comptes de tenir un únic valor per p constant, RED li assigna **un valor diferent per a cada datagrama!**

El valor de p dependrà de la relació entre la mida de la cua i els límits T_{min} i T_{max} .



Encara que aquest esquema linal és la base del càlcul de probabilitats de RED, cal fer una **modificació** per a poder utilitzar-lo.

→ Si l'apliquèssim tal qual, la probabilitat d'eliminar els darrers segments d'una ràfaga seria alta.

Seria interessant no perdre els darrers de les **ràfagues**, però sí els que acaben d'omplir la cua... i tot al mateix temps!

Idea:

Podem fer com a TCP. Es pot mantenir un **promig** de la mida de la cua per a calcular les probabilitats!

Calculem l'estimació de la mida:

$$\text{mida} = (1 - \gamma) * \text{estimació} + \gamma * \text{mida cua}, \quad 0 \leq \gamma \leq 1$$

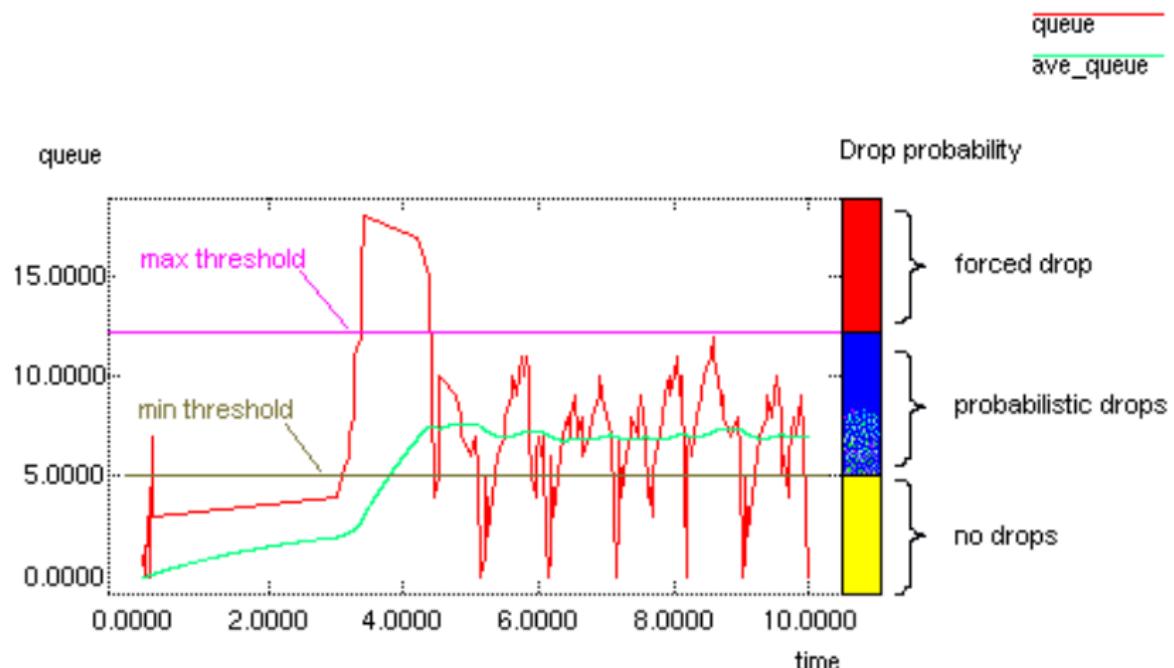
Per a γ petita, el promig reflexa els canvis que duren, però és inmune a les ràfagues.

Detall d'implementació

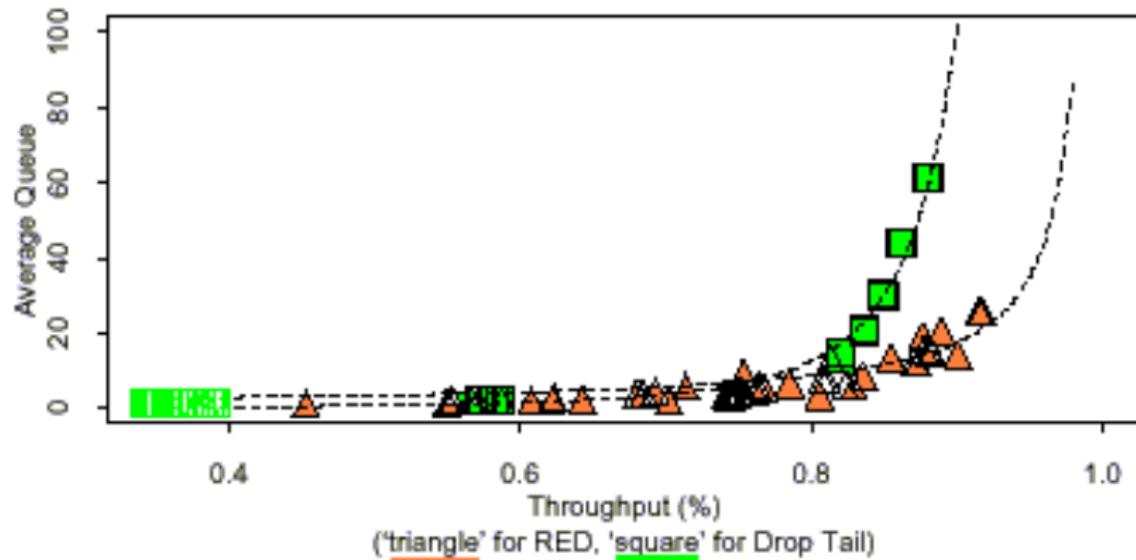
La mida de la cua no és en número de datagrames, sino que es fa per bytes!

- Els datagrames grans tindran més probabilitats de ser descartats que els petits (no hi haurà espai).
- En treballar per bytes, les confirmacions tindran menys probabilitats de perdre's en un camí congestionat.

Exemple de funcionament d'una política RED:



Comparació entre RED i Tail Drop:



Contingut

- 1 Repàs TCP
- 2 Timeouts i retransmissions
- 3 Resposta a la congestió
 - Introducció
 - Cues
 - Tail Drop
 - RED
 - Altres cues
- 4 Finestres menguants
- 5 Millores de TCP i rendiment

La recomanació de l'[IETF](#) és fer servir la política **RED**.

Tot i així, hi ha altres polítiques de gestió de cues que poden ser més útils en alguns casos:

- pfifo_fast (**FIFO**)
- Stochastic Fairness Queuing (**SFQ**)
- Token Bucket Filter (**TBF**)
- Differentiated Services Mark (**DSMark**)
- Class Based Queuing (**CBQ**)
- Hierarchy Token Bucket (**HTB**)

Veiem a continuació les seves principals característiques.

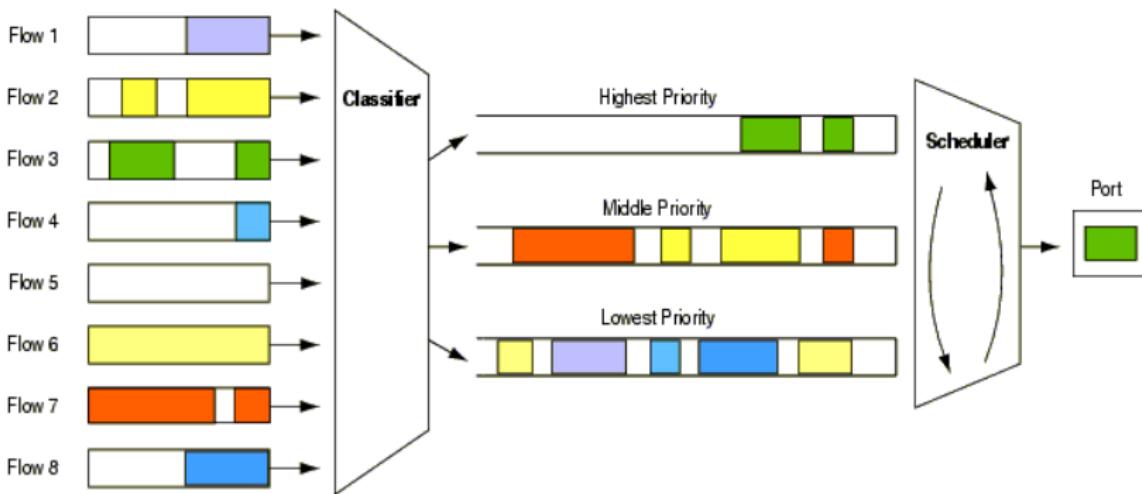
pfifo_fast (FIFOF)

És igual que la política de *pfifo* (*Tail Drop*), però utilitzant **tres bandes** diferents. El datagrama va a una banda o a una altra dependent del seu valor de **TOS**.



Primer es processen els datagrames de les bandes més **prioritaries**. De fet, la FIFOF és un cas concret de priorització de cues (PRIO).

Esquema de la política de gestió de cues FIFO:



Stochastic Fairness Queuing (SFQ)

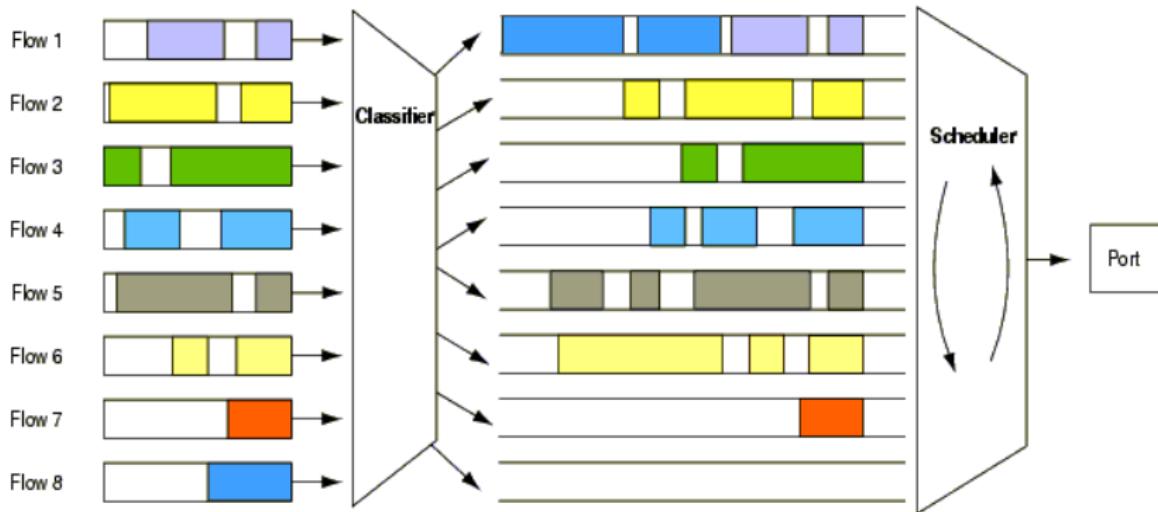
Cada “conversa” es manté en una cua FIFO diferent. Les converses són identificades per les adreces IP i els ports.

Es van agafant datagrames d'una cua o d'una altra seguint un esquema *round-robin*.

Per a no tenir un nombre de cues massa gran s'utilitza un mecanisme de *hash*, que cal anar re-calculant. D'aquí el nom d'estocàstic.



Esquema de la política de gestió de cues SFQ:



Token Bucket Filter (TBF)

Tenim una “galleda” (*bucket*) que es va omplint amb “peces” (*tokens*) a una certa velocitat. Cada byte que arriba a la cua pren un token i passa.

- Si les dades arriben a la cua més a poc a poc que els tokens, aquests s’acumulen i permetran ràfagues curtes per sobre de la freqüència d’arribada de tokens.
- Si les dades arriben més ràpides, la velocitat de sortida de la cua és com la d’entrada de tokens.
- Si el bucket està ple i arriben més dades, s’eliminen.



Differentiated Services Mark (DSMark)

Utilitza els serveis difenciats per a ofereir diferents cues segons la qualitat de servei requerida. (Diferent al Type of Service).

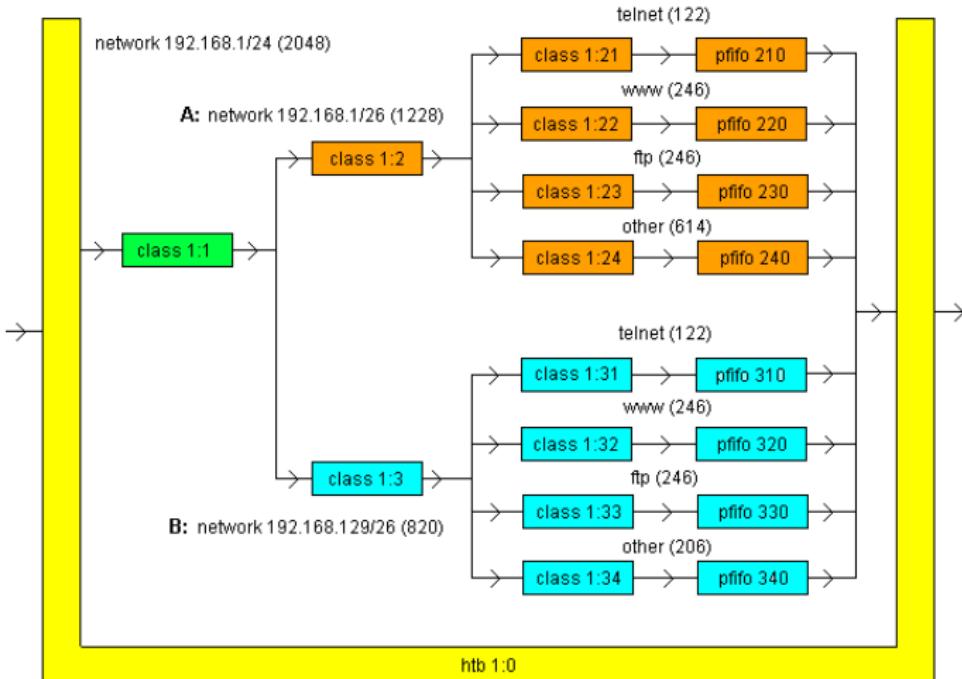
Class Based Queuing (CBQ)

Classifica els datagrames en classes i assigna una disciplina de cua a cadascuna.

Hierarchy Token Bucket (HTB)

Divideix el tràfic per diferents sub-interfícies virtuals, i limita l'amplada de banda de cadascuna amb una política de TBF.

Esquema de la política de gestió de cues HTB.



Contingut

1 Repàs TCP

2 Timeouts i retransmissions

3 Resposta a la congestió

4 Finestres menguants

- El problema de les finestres menguants
- Solucions al SWS

5 Millores de TCP i rendiment

Finestres menguants

El TCP segueix el mecanisme clàssic de **finestres lliscants** (*sliding windows*), amb algunes particularitats (p.e. es fa sobre els **bytes**, no sobre els segments).



A part de les particularitats que vam veure a Xarxes II, hi ha d'altres més específiques sobre el funcionament.

Detalls funcionament finestres lliscants

- **Compromís de finestra:** TCP no hauria de fer més petita la finestra de la mida anunciada prèviament.
Els anuncis haurien de ser més petits en les confirmacions, per a que canviï al mateix temps que la finestra avança.
- **Fre d'emissió:** Si la finestra anunciada pel receptor és zero, l'emissor deixa d'enviar.
- **Restabliment emissió:** Quan el receptor torna a poder rebre dades envia un anuncí diferent de zero.

Excepcions

Hi ha dues excepcions d'enviament amb finestra zero:

- L'emissor pot emetre encara que l'anunci sigui de zero si hi ha **dades urgents** preparades.
- De tant en tant es va interrogant al receptor per a evitar entrar en un **deadlock** (**timer de persistència**).

Aquest timer usa un *backoff* exponencial de 1.5 segons a 60, però prenent com a mínim 5 (i.e. **5,5,6,12,24,48,60**). Mai acaba.



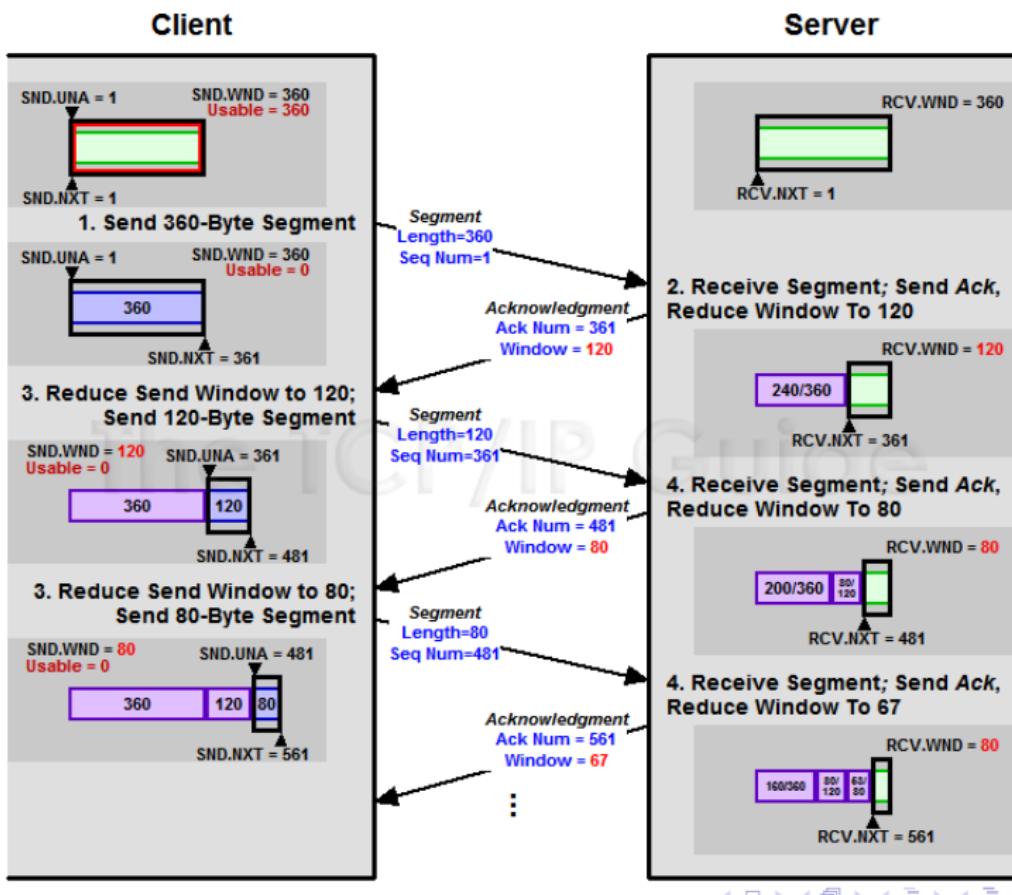
El síndrome de la finestra tonta (*Silly Window Syndrome*)

- ➊ El receptor prepara un buffer de K bytes per a rebre i envia un **anunci** de finestra amb aquesta mida.
- ➋ L'emissor envia aquests bytes i el receptor respondrà eventualment amb la confirmació dels bytes. Com que ja tindrà el buffer ple, l'anunci de finestra que envia és de **zero bytes**.
- ➌ L'aplicació en el receptor llegirà pocs bytes del buffer, deixant poques posició lliure, i TCP enviarà un **anunci de pocs bytes**.
- ➍ L'emissor enviarà pocs bytes. El receptor confirmarà, l'aplicació alliberarà unes posicions més, s'incrementa l'anunci de finestra, **s'envien pocs bytes**, ... i el cicle s'anirà repetint per sempre.

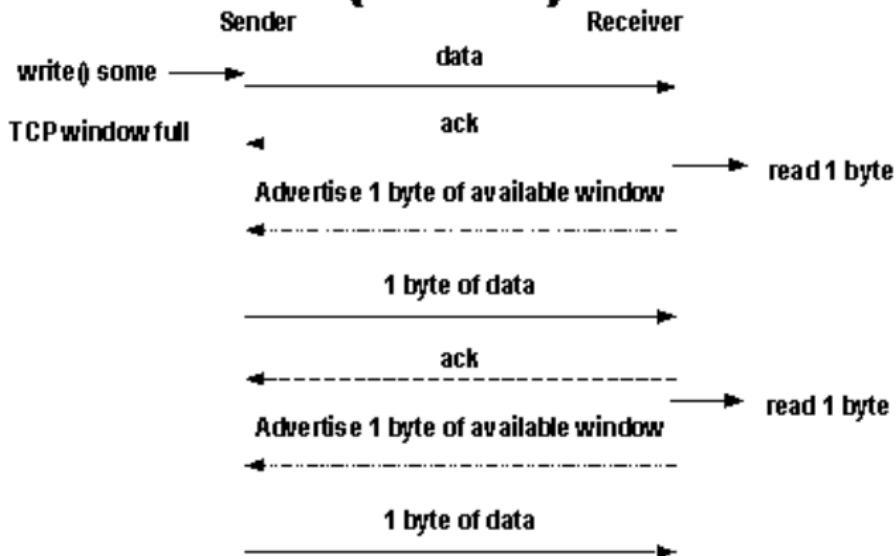
Això és molt dolent! Pocs bytes de dades per 20 bytes de capçaleres de TCP, més 20 bytes de capçaleres d'IP (sense comptar opcions).

- Consum d'**amplada de banda**
- **Ratio gran** de capçalera/dades
- **Overhead** de càlcul (procès, checksums, formació segment, ...)
- **Temps** més gran de transmissió





The Silly Window Syndrome (SWS)



Overhead: 3 packets and $3 \times 40 = 120$ bytes of TCP/IP headers transport 1 byte of data!

Aquesta situació també podria produir-la **l'emissor**, si l'aplicació li passa dades d'un byte en un byte.

→ Aquest problema afectava les primeres implementacions de TCP.



Cóm podem evitar-ho?

Tant l'emissor com el receptor han d'implementar tècniques per a evitar el **SWS**. Aquestes tècniques estaran basades en heurístiques que previndran enviar poques dades en cada segment, i enviar increments petits de finestra.

Contingut

1 Repàs TCP

2 Timeouts i retransmissions

3 Resposta a la congestió

4 Finestres menguants

- El problema de les finestres menguants
- Solucions al SWS

5 Millores de TCP i rendiment

Necessitem aplicar tècniques en els dos extrems (emissor i receptor), ja que tenim una connexió *full-duplex*!

→ Veurem 2 tècniques des del punt de vista del **receptor** de dades i 1 des de la de **l'emissor**.

Receptor



Receptor: 1a tècnica (de D. Clark)

Una primera tècnica consisteix en **esperar** enviar l'anunci de finestra (després de que sigui zero) a que l'increment sigui **prou gran**.

Però, i quant és “**prou gran**”?

El llindar per a considerar que l'increment és prou gran depén de la mida del buffer i del MSS:

$$\text{Llindar} = \min\left(\frac{\text{mida buffer}}{2}, \text{Dades en MSS}\right)$$

→ Això evita segments amb poques dades si l'aplicació receptora consumeix molt a poc a poc.

Receptor: 2a tècnica

Una segona tècnica consisteix en **no enviar les confirmacions fins que l'anunci de finestra sigui prou gran**.

→ L'estàndard **recomana** utilitzar aquesta tècnica.

Aquesta segona tècnica té pros i contres:

- **Principal avantatge:** Decrementa el tràfic de segments i augmenta el *throughput* del protocol.

En el cas anterior necessitavem una confirmació sense increment de finestra, i un altre per anunciar l'ampliació.

- **Principal desavantatge:** Si esperem massa tindrem retransmissions!!

La retransmissió ocasiona ús innecessari de l'amplada de banda i, per tant, fa baixar el *throughput*.

L'estimació del RTT resultarà també afectada, baixant la velocitat de transmissió!!

→ Per a buscar un **compromís**, les implementacions marquen un **límit** pel temps d'endarreriment de les confirmacions:

màx. delay = 500 ms.

I, per a garantitzar bones estimacions del RTT, s'envia una confirmació a **cada segment alternat**.

Emissor

Per tal de no generar un tràfic de segments petits, l'emissor evita enviar el segment fins que tingui una quantitat raonable de dades:

múltiples *write* → 1 segment

D'aquesta tècnica s'en diu *clumping*.

→ Problemes: Si esperem molt la comunicació es relentitza. Si esperem poc els segments seran petits.

Fixar un temps (com a les aplicacions de terminal remot) no és una solució óptima en general. Volem una funció adaptativa.

Cóm ho fem? Enviem les dades... quan rebem la confirmació anterior!

Algorisme de Nagle:

```
si hi ha dades per enviar
    si mida finestra i dades disponibles >= MSS
        Enviar un segment amb MSS bytes
    sino
        si queden dades no confirmades al buffer
            posar dades en cua fins que arribi ACK
        sino
            enviar dades immediatament
```

L'algorisme de Nagle és adaptatiu:

- En una transferència de fitxers (aplicació ràpida en comparació a la xarxa) els segments successius tindran molts bytes.
- En un tecleig de comandes remotes (aplicació lenta respecte la xarxa) s'envien petits segments sense espera.

Resulta molt senzill **d'implementar**. No tenim temporitzadors separats per cada connexió. Tampoc no treu capacitat als casos més habituals.

→ És **requerit** a l'estàndard de TCP.

Contingut

- 1 Repàs TCP
- 2 Timeouts i retransmissions
- 3 Resposta a la congestió
- 4 Finestres menguants
- 5 Millores de TCP i rendiment
 - Millores de TCP
 - Rendiment

Millores de TCP

TCP ha funcionat bé des de fa molts anys en xarxes que van dels 1200 bits/seg fins a Gigabit Ethernets. Les xarxes **ràpides** actuals, **sense fils** i les **LFN (Long Fat Networks)**, però, mostren **alguns límits** del protocol.



Algunes propostes per superar aquests límits són:

- Mecanisme de **descobriment de MTU** del camí
- Opció **d'escalat** de la finestra
- Timestamping pel calcul precís del RTT
- Transaction TCP (**T/TCP**)
- Confirmacions selectives

Aquests mecanismes són **retrocompatibles**.

Un problema a destacar en xarxes d'alta capacitat és la **re-utilització de números de seqüència**.

→ Què passa si un segment **re-apareix** més tard, quan la seva connexió ja ha acabat i hi ha una altra amb els mateixos *end points*?

En principi els datagrames tenen una vida màxima de 255 segons (o *hops*, el que passi abans), i els segments de 120 segons. Això evita que passi (a Ethernet necessitem 60 minuts per re-utilitzar números!), però en xarxes ràpides **encara hi ha risc**.

Fins i tot, en una xarxa ràpida, podriem trobar el mateix número de seqüència en la mateixa connexió!, dintre dels 120 segons de vida del segment (tots els números poden utilitzar-se en 34 segons en una xarxa Gigabit Ethernet).

PAWS

Una manera de solucionar aquest problema és utilitzar l'algorisme **PAWS** (*Protection Against Wrapped Sequence Numbers*).

L'algorisme bàsicament utilitza una opció de *timestamp* i considera el número de seqüència com la **concatenació** entre el número de seqüència en la capçalera concatenat amb aquest *timestamp*.

Contingut

- 1 Repàs TCP
- 2 Timeouts i retransmissions
- 3 Resposta a la congestió
- 4 Finestres menguants
- 5 Millores de TCP i rendiment
 - Millores de TCP
 - Rendiment

Rendiment de TCP

El *throughput* del protocol en una xarxa el podem mesurar així:

$$\text{Throughput(bits)} = \frac{\text{ampladaBanda(bits/seg)} * \text{dadesUtils}}{\text{dadesEnviades}}$$

→ Per a calcular el ratio de dadesUtils i enviades hem de tenir en compte tots els bits que s'envien **realment**.

Per exemple, en una Ethernet a 10Mbit/seg fariem els càlculs següents.

Bytes d'un segment de dades i de confirmació enviats sobre una xarxa Ethernet 10Mb:

Camp	Dades	ACK
Preamble Ethernet	8	8
Adreces Ethernet	12	12
Camp tipus Ethernet	2	2
Capçalera IP i TCP	40	40
Dades usuari	1460	0
Padding ethernet	0	6
CRC Ethernet	4	4
Espai entre paquets ($9.6\mu s$)	12	12
Totals	1538	84

Suposem que enviem dos segments de dades plens i una confirmació.

$$\begin{aligned} \text{Throughput} &= \frac{10000000 \text{ bits/seg}}{8 \text{ bits/byte}} * \frac{2*1460 \text{ bytes}}{2*1538+84 \text{ bytes}} = \\ &= 1155063 \text{ bytes/seg} \end{aligned}$$

Si la finestra de TCP s'obre al màxim (65535) permetria tenir una finestra de 44 segments de 1460 bytes. Si el receptor envia una confirmació cada 22 segments, el *throughput* teòric quedaría així:

$$\begin{aligned} \text{Throughput} &= \frac{10000000 \text{ bits/seg}}{8 \text{ bits/byte}} * \frac{44*1460 \text{ bytes}}{44*1538+2*84 \text{ bytes}} = \\ &= 1183667 \text{ bytes/seg} \end{aligned}$$

Cal tenir en compte alguns límits pràctics en el món real:

- No es pot anar més depresa que l'enllaç més lent.
- No es pot anar més depresa que l'amplada de banda de la memòria de la màquina més lenta.

N'hi han moltes altres restriccions que depenen de xarxes concretes.