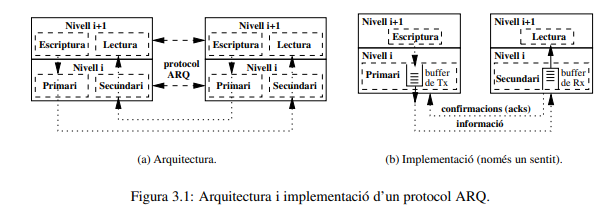
**Tema 4: Protocols punt-a-punt, el protocol TCP**

L’objectiu d’aquest tema és l’estudi del Transmission Control Protocol (TCP) d’Internet. TCP és un dels protocols amb més complexitat que es fa servir a Internet. Per això, abans d’abordar el seu estudi, veurem alguns algorismes bàsics relacionats amb TCP: Els algorismes Automatic Repeat reQuest (**ARQ**). Els algorismes ARQ els podem trobar en protocols que, com TCP, estan distribuïts entre dues entitats o “punts”, d’aquí el seu nom de protocols “punt-a-punt”. Aquest nom es fa servir per contraposició a altres algorismes distribuïts en més d’un punt, com ara els algorismes multiaccés que es fan servir en les LANs.



**3.1 Protocols ARQ bàsics**

L’objectiu dels protocols ARQ és aconseguir un canal de comunicació fiable, és a dir, que la informació transmesa arribi sense errors, sense duplicacions i en el mateix ordre en què s’envia. La figura 3.1 mostra els ingredients d’un protocol ARQ. Típicament, desitjarem que el canal sigui **bidireccional**, vegeu la figura 3.1(a): en cada un dels nivells que implementen el protocol hi ha una entitat que transmet informació (que anomenarem “**primari**”) i una que la consumeix (que anomenarem “**secundari**”). El principi de funcionament dels protocols ARQ es basa en la retransmissió de la informació que no arribi o que arribi amb errors al secundari. Perquè el primari sàpiga si la informació ha arribat correctament, el secundari envia confirmacions (acknowledgments, o acks per abreujar) al primari. La figura 3.1(b) mostra els elements bàsics de la implementació d’un protocol ARQ. En el primari hi ha un “buffer de transmissió” on es guarda tota la informació que s’ha enviat i que encara no ha estat confirmada pel secundari. A mesura que hi ha espai lliure en el buffer de transmissió, el primari deixa que el nivell superior hi escrigui, per tenir informació llesta per enviar al secundari. En cas d’error, el primari pot retransmetre la informació perquè la té emmagatzemada en el buffer de transmissió. A mesura que arriben confirmacions del secundari, el primari esborra la informació confirmada del buffer de transmissió. El secundari té un buffer de recepció on guarda la informació rebuda fins que la llegeix el nivell superior.

Altres ingredients que formen part dels algorismes ARQ són els següents:

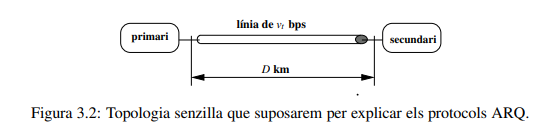
* Números de seqüència: com veurem més endavant, per poder relacionar els missatges d’informació i les seves corresponents confirmacions, cal fer servir “números de seqüència”.
* Protocol orientat a la connexió (connection oriented): es diu que un protocol és orientat a la connexió quan té una fase inicial “d’establiment de la connexió” i una fase final de “terminació”. En aquestes fases s’envien missatges de senyalització que serveixen per reservar recursos i deixar el protocol en una fase inicial coneguda. Si aquestes fases no hi són, es diu que el protocol és “no orientat a la connexió” (connectionless). Durant l’explicació que farem dels protocols ARQ, suposarem que la connexió ja està establerta.

Hi ha tres algorismes ARQ bàsics, per ordre de complexitat són:

* Stop and wait.
* Go back N.
* Retransmissió selectiva (Selective retransmission).

El nivell que implementa el protocol ARQ no té per què ser el de transport (com és el cas de TCP). En la pràctica podem trobar protocols d’altres nivells (típicament d’enllaç) que implementen un algorisme ARQ. Així doncs, farem servir el terme genèric de PDU (Protocol Data Unit) per referir-nos als missatges intercanviats pel protocol. En concret, per analitzar els algorismes ARQ bàsics anteriors, farem servir la següent notació: amb IK ens referirem a la PDU d’informació *k* enviada pel primari, i amb AK ens referirem a la seva confirmació.

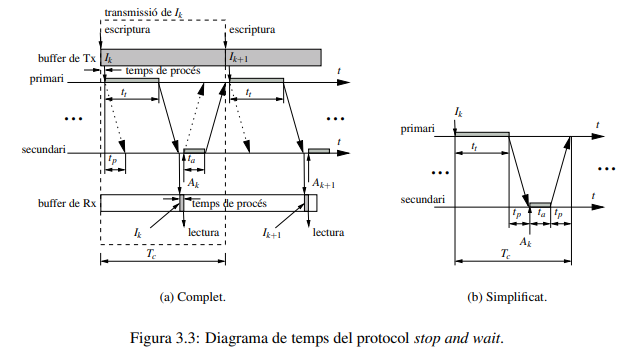
**3.1.1 Stop and wait**



El principi de funcionament d’stop and wait és: “Transmetre una PDU d’informació i esperar que es confirmi abans de transmetre’n una de nova”. Hi ha diverses maneres d’implementar aquest principi. Nosaltres suposarem l’algorisme que es detalla a continuació. Per explicar-ho, suposarem la topologia senzilla de la figura 3.2 i ens ajudarem del diagrama de temps de la figura 3.3. La topologia està formada per un primari i un secundari connectats per un cable de *D* km de distància i velocitat de transmissió de vt bps. La velocitat de propagació del senyal elèctric en el cable és de vp m/s. Els events que mostra la figura 3.3(a) són els següents:

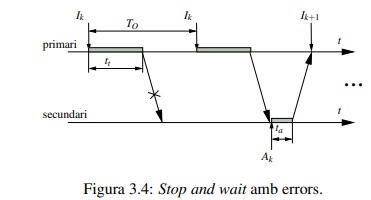
* El nivell superior escriu la informació que s’ha de transmetre. En un temps de procés que suposarem 0, el primari “ensambla” la PDU d’informació IK de Lt bits amb aquesta informació, la guarda en el buffer de transmissió i la passa al nivell inferior per a la seva transmissió.
* La transmissió comença immediatament i dura un temps tt[segons] = Lt[bits]/vt[bps].
* Els temps de propagació de cada bit que es transmet dura tp[segons] = D[m]/vp[m/s]. Si és el buit, vp ≈ 3×108 [m/s]. En un conductor, vp ≈ 2×108 [m/s].
* Quan arriba l’últim bit de IK al secundari, el nivell superior llegeix la informació rebuda en un “temps de procés” que suposarem igual a 0 i el secundari envia la confirmació Ak. Si la mida de la confirmació és de La bits, el temps de transmissió de la confirmació serà de ta[segons] = La[bits]/vt[bps]. Normalment, la mida de la PDU amb les confirmacions (La) és molt menor que la de les PDUs d’informació (Lt) per tant, ta < tt .
* Els bits de la confirmació tarden un temps tp a arribar al primari. Quan arriba l’últim bit de la confirmació, el primari esborra IK del buffer de transmissió i repeteix el procés per a una nova PDU (Ik+1).

Per representar el diagrama de temps de la figura 3.3(a) farem servir el diagrama simplificat de la figura 3.3(b): és a dir, representarem només els instants en què es comença i acaba de transmetre la PDU, i els instants on es rep l’últim bit de la PDU.



3.1.1.1 Retransmissions amb stop and wait:

En cas d’error suposarem el diagrama de temps de la figura 3.4. Cada vegada que el primari envia una PDU, activa un temporitzador (TO en la figura). Si el temporitzador “salta” sense haver rebut la confirmació (en anglès es diu que es produeix un time-out), aleshores el primari retransmet la PDU. D’aquesta manera, si la PDU IK es perd (no arriba al secundari), o arriba amb errors (i el secundari la descarta), saltarà el time-out i el primari retransmetrà la PDU.

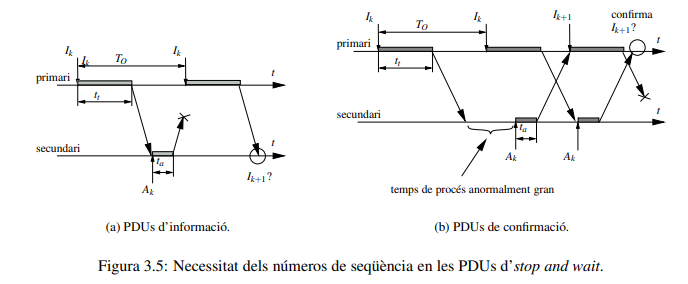


3.1.1.2 Necessitat dels números de seqüència:

Tal com s’ha introduït en la secció 3.1, els protocols ARQ necessiten un número de seqüència per poder relacionar les PDUs d’informació i les confirmacions. Aquest número de seqüencia en la pràctica és un dels camps de la capçalera que afegeix el protocol. A continuació veurem un exemple que intenta justificar la necessitat d’afegir el número de seqüència tant en les PDUs d’informació com en les confirmacions.

**PDUs d’informació**: considerem l’exemple de la figura 3.5(a). La confirmació es perd i el primari retransmet la PDU IK. Si la PDU no portés número de seqüència, el secundari no tindria manera de detectar la recepció duplicada de la PDU IK.

**PDUs de confirmació**: considerem l’exemple de la figura 3.5(b). Ara suposem que el temps de procés del secundari no és zero. Per contra, suposem que és anormalment gran, i el primari retransmet la PDU IK abans de rebre la confirmació AK. Això podria passar, per exemple, si el secundari és una estació multitasca molt carregada, de forma que el procés que llegeix el secundari només rep atenció després d’un temps significatiu. D’aquesta manera, el nivell que hi ha per sobre del secundari no llegeix les PDUs immediatament després d’estar disponibles, sinó quan ha passat un cert temps. Si les confirmacions no portessin número de seqüència, el primari no **detectaria la recepció de la confirmació duplicada**. Per exemple, en la figura 3.5(b) el primari podria interpretar que es confirma IK+1 i transmetre IK+2, de manera que IK+1 es perdria.

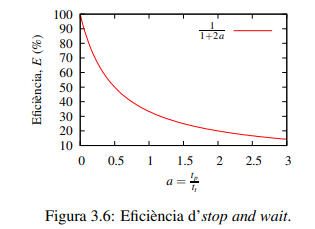


3.1.1.3 Eficiència del protocol stop and wait:

Considerem la figura 3.3(b). Calculant l’eficiència del protocol com el temps que transmet una PDU d’informació (tt en la figura), respecte el temps total que es necessita com a mínim per transmetre-la (Tc en la figura), tenim que:



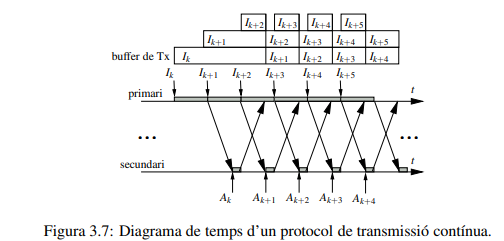
Observem que s’ha fet l’aproximació que el temps de transmissió de les PDUs de confirmació és molt menor que el temps de transmissió de les PDUs d’informació. La figura 3.6 mostra l’eficiència del protocol en funció del paràmetre *a = tp/tt* . La conclusió és que si *a* és molt petit (tp és molt més petit que tt), aleshores l’eficiència és pròxima al 100 %. Altrament, l’eficiència decreix ràpidament (quan tp = tt -> E ≈ 30 %).



**3.1.2 Protocols de transmissió contínua**

En la secció anterior hem vist que el protocol stop and wait pot ser molt ineficient si no es compleix que el temps de propagació (tp) és molt més petit que el temps de transmissió de les PDUs (tt). Els protocols de transmissió contínua resolen aquest problema deixant que el primari enviï més d’una PDU sense confirmar. La figura 3.7 mostra el funcionament de transmissió contínua. Cada vegada que es transmet una PDU d’informació, es guarda en el buffer de transmissió (per si s’ha de retransmetre). Quan es rep la corresponent confirmació, s’esborra del buffer. En absència d’errors l’eficiència del protocol (sense tenir en compte les capçaleres) és del 100 %. Els algorismes bàsics de recuperació d’errors en una transmissió contínua són:

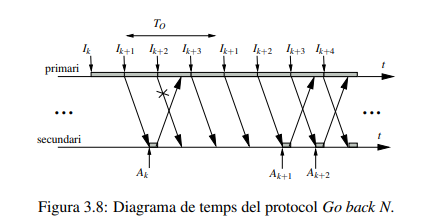
Go back N i retransmissió selectiva.



3.1.2.1 Go back N:

La idea del protocol Go back N és que en cas d’error el primari “torni enrere” fins la PDU que falta al secundari i comenci a transmetre novament a partir d’aquest punt. Hi ha diverses maneres d’implementar aquest principi. Nosaltres suposarem l’algorisme que es detalla a continuació (vegeu la figura 3.8):

* Les confirmacions són acumulatives, és a dir, la confirmació Ak confirma totes les PDU d’informació amb números de seqüència ≤ k.
* Si el secundari rep una PDU d’informació (IK) amb errors o fora de seqüència:
  + (i) Deixa d’enviar confirmacions fins que rep correctament la PDU que falta.
  + (ii) Descarta totes les PDUs que rep amb número de seqüència diferent de k.
* Quan salta el temporitzador de retransmissió d’una PDU (IK), el primari retransmet la PDU IK, i contínua amb la transmissió de Ik+1 Ik+2, ...

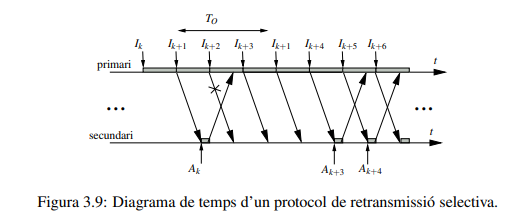


3.1.2.2 Retransmissió selectiva:

La idea d’un protocol de retransmissió selectiva és que el secundari no descarti mai les PDUs que arriben correctament, encara que arribin fora de seqüència (a diferència del que fa Go back N). Això permet millorar l’eficiència de Go back N en cas d’error, a costa de complicar la implementació del secundari: amb retransmissió selectiva haurà d’emmagatzemar i ordenar les PDUs que arriben fora de seqüència. És a dir, com que en Go back N el secundari sempre rep les PDU en seqüència (si en falta alguna, descarta totes les que arriben fins que arriba la que falta), el buffer de recepció del secundari sempre tindrà les PDUs rebudes en seqüència, llestes per ser llegides pel nivell superior. Amb retransmissió selectiva, en canvi, pot haver-hi PDUs en el buffer de recepció que no es poden lliurar al nivell superior perquè en falta alguna de número seqüència anterior que encara no ha arribat (o ho ha fet amb errors i s’ha hagut de descartar).

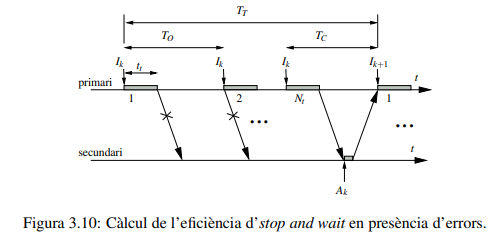
Hi ha diverses maneres d’implementar un protocol de retransmissió selectiva. Nosaltres suposarem l’algorisme que es detalla a continuació (vegeu la figura 3.9):

* Les confirmacions són acumulatives, és a dir, la confirmació AK confirma totes les PDU d’informació amb números de seqüència ≤ k.
* Si el secundari rep una PDU d’informació (IK) amb errors o fora de seqüència:
  + (i) Deixa d’enviar confirmacions fins que rep correctament la PDU que falta.
  + (ii) Guarda totes les PDUs que rep amb número de seqüència diferent de k.
* Quan salta el temporitzador de retransmissió d’una PDU (IK), el primari retransmet la PDU IK, però no retransmet altres PDUs que ja havia enviat abans. És a dir, només retransmet les PDUs per les quals salta el time-out.
* Quan el secundari rep una retransmissió, envia una confirmació acumulada que confirma fins a l’última PDU en seqüència rebuda correctament.



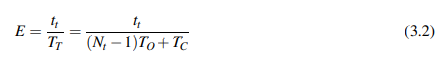
**3.1.3 Eficiència en presència d’errors**

En aquesta secció comparem els protocols ARQ bàsics que hem vist anteriorment quan la probabilitat de pèrdua o error d’una PDU és diferent de zero. Per caracteritzar aquesta probabilitat, suposarem que coneixem la mitjana del nombre de transmissions necessàries per a la transmissió amb èxit d’una PDU (Nt). És a dir, cada PDU es transmet en mitjana Nt − 1 vegades sense èxit, i la transmissió Nt arriba correctament.

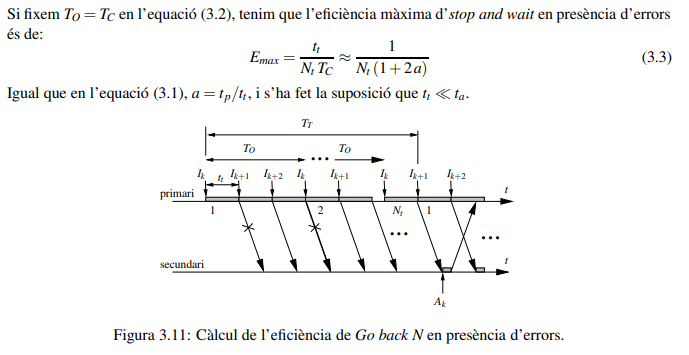


Stop and wait:

La figura 3.10 mostra el diagrama de temps representatiu d’una transmissió amb errors fent servir stop and wait. Per calcular l’eficiència, podem suposar que les Nt transmissions que apareixen en la figura es repeteixen de forma periòdica. Durant aquest temps només es transmet una PDU d’informació. És a dir, s’aprofita un temps de transmissió d’una PDU d’informació (tt en la figura) per cada temps que duren les Nt transmissions (TT en la figura). En les Nt −1 primeres retransmissions la PDU es retransmet després d’un temps TO (quan salta el temporitzador de retransmissió), mentre que la PDU IK+1 es transmet en un temps TC després de la retransmissió Nt (vegeu la figura). Per tant, el temps invertit en la transmissió de la PDU IK és de: TT = (Nt −1)TO +TC. L’eficiència és, doncs:

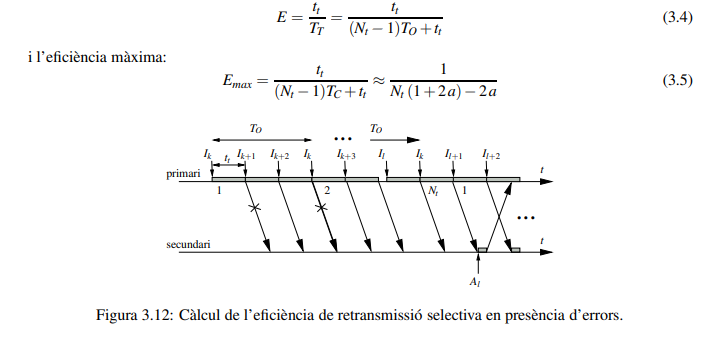


Per no tenir retransmissions innecessàries, com que el temporitzador de retransmissió salta abans de rebre la confirmació, ha de complir-se: TO > TC. Si els retards són constants, aleshores l’eficiència màxima es pot aconseguir quan TO > TC. En la pràctica, però, els retards, i per tant TC, són variables. Així doncs, si TC és la mitjana, s’ha de deixar un temps de “guarda” (δ) i fixar un valor de TO = TC +δ.



Go back N:

La figura 3.11 mostra el diagrama de temps representatiu d’una transmissió amb errors fent servir Go back N. Raonant igual que abans tenim:

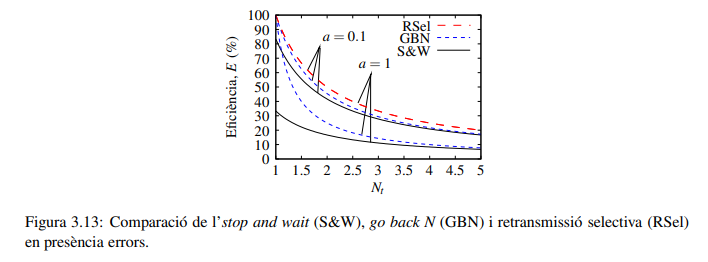


Retransmissió selectiva:

La figura 3.12 mostra el diagrama de temps representatiu d’una transmissió amb errors fent servir retransmissió selectiva. A diferència d’stop and wait i Go back N, el temps que queda lliure mentre es fan les Nt transmissions d’una mateixa PDU d’informació s’aprofita per transmetre altres PDUs. Així doncs, el temps invertit per transmetre una PDU d’informació és de Nt\*tt i l’eficiència és independent del temporitzador de retransmissió, i val:



Comparació:



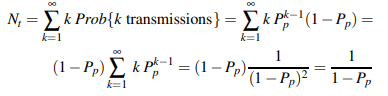
La figura 3.13 mostra l’eficiència dels tres protocols anteriors en funció de Nt per a dos valors de *a*. La conclusió és que per *a* < 1, el comportament dels tres protocols és semblant. Si això no es compleix, stop and wait té una eficiència baixa, independentment de la probabilitat d’error. En canvi, Go Back N i retransmissió selectiva tenen una E ≈ 100% per una probabilitat d’error petita Nt ≈ 1, retransmissió selectiva té una eficiència bastant millor que go back N per a una probabilitat d’error no molt alta, i per a probabilitat d’error altes, els tres protocols tenen una eficiència semblant.

Càlcul de Nt:

En els apartats anterior hem fet servir la mitjana del nombre de transmissions necessàries per a la transmissió amb èxit d’una PDU (Nt). Normalment, el paràmetre que és més fàcil de conèixer és la probabilitat d’error en el bit Pb. Ara calcularem la relació que hi ha entre aquest dos paràmetres. Perquè el càlcul sigui senzill, suposarem que cada bit de la PDU té una probabilitat d’error independent dels altres bits i igual a Pb. En aquest cas, si la PDU té L bits, la probabilitat que la PDU tingui algun error (Pp) val:



La probabilitat d’haver de transmetre una PDU k vegades és la probabilitat de les k−1 primeres vegades tingui error (Ppk-1 ) multiplicat per la probabilitat que l’última transmissió no tingui error (1 – Pp). Per tant:



**3.1.4 Control de flux**

Els protocols ARQ que hem vist en les seccions anteriors no només s’utilitzen per a la recuperació d’errors: un dels seus objectius és el control de flux. L’objectiu del control de flux és l’adaptació de la velocitat de transmissió eficaç entre el transmissor (el primari en un protocol ARQ) i el consumidor (el secundari en un protocol ARQ).

Veiem un exemple: el control de flux entre un PC i un mòdem connectat al port sèrie. El port sèrie fa servir el protocol de nivell físic RS232. Aquest protocol té dues línies que serveixen per al control de flux: Request To Send (RTS) i Clear To Send (CTS). El funcionament d’aquestes línies és el següent:

Quan el PC està llest per transmetre dades, activa la línia RTS. Si el mòdem les pot rebre, activa la línia CTS i el PC les transmet. Si la línia CTS està desactivada, el PC no pot transmetre. La línia sèrie es configura amb una velocitat de transmissió major de la que pot aconseguir el mòdem a través de la línia telefònica. El mòdem té un buffer de transmissió on guarda la informació que transmet a través de la línia telefònica. Quan el mòdem activa la línia CTS, el PC li envia informació a una velocitat major de la que pot enviar el mòdem a la línia telefònica. Així doncs, el buffer de transmissió del mòdem s’omple. Quan arriba a un cert llindar, el mòdem desactiva la línia CTS i torna a activar-la quan el buffer es comença a buidar. D’aquesta manera el mòdem sempre té informació llesta per transmetre a través de la línia telefònica i podrà aprofitar al màxim la línia. Per altra banda, el control de flux de les línies RTS/CTS permet ajustar la velocitat eficaç de la línia sèrie a la de la línia telefònica, sense que es perdi informació en el mòdem.

Suposem ara un protocol ARQ entre dos PCs d’una mateixa xarxa. Si un dels PCs envia informació a una velocitat eficaç major de la que pot consumir el receptor (per exemple, perquè el receptor és un PC més lent i la velocitat de transmissió de la xarxa és elevada), aleshores arribarà un moment que el buffer de recepció del receptor vessarà i es perdrà la informació. Aquesta situació és indesitjable, per això, cal un mecanisme de control de flux. El protocol stop and wait permet un control de flux inherent al seu funcionament: si el secundari augmenta el temps en enviar les confirmacions, la velocitat eficaç disminuirà i s’ajustarà al secundari. En el cas dels protocol de transmissió contínua cal introduir alguna condició addicional per aconseguir el mateix: la finestra de transmissió.

3.1.4.1 Protocols de finestra:

En un protocol de transmissió contínua el primari pot enviar més d’una PDU sense confirmar. Per poder fer un control de flux cal introduir un límit al nombre de PDUs que pot enviar: Aquest límit s’anomena “finestra de transmissió”. El funcionament és el següent:

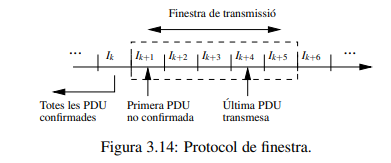
* El primari pot enviar fins a W PDUs d’informació sense confirmar.
* El secundari només envia la confirmació de la PDU d’informació IK després que el nivell superior hagi llegir aquesta PDU.

La figura 3.14 il·lustra el funcionament d’un protocol de finestra. Suposarem que les confirmacions són acumulatives, tal com ho hem fet en Go back N i retransmissió selectiva. Amb aquesta suposició, totes les PDU amb número de seqüència menor o igual a l’última confirmada estan confirmades i el primari les ha esborrat del buffer de transmissió.

En aquest exemple la finestra val W = 5. El primari transmet tan aviat com ho permet el nivell inferior. Abans de cada transmissió, però, comprova que la diferència entre el número de seqüència de la PDU a transmetre i el de l’última PDU confirmada sigui inferior a la finestra. Si no és així el primari es queda bloquejat fins que arriben noves confirmacions. En l’exemple, si no arriben més confirmacions el primari es quedaria bloquejat després de transmetre la PDU IK+5. Quan arriben confirmacions de noves PDUs, l’índex que apunta a l’última PDU confirmada, i per tant la finestra de PDUs que es poden transmetre, “avancen” i permeten la transmissió de noves PDUs.

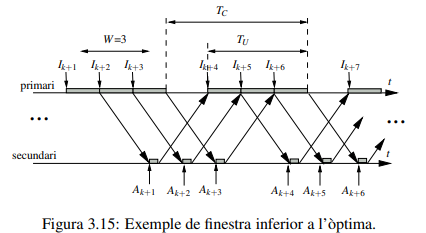
De l’explicació anterior podem deduir que el protocol stop and wait és equivalent a un protocol de finestra amb una mida de la finestra igual a 1: després de la transmissió d’una PDU, el primari es queda bloquejat fins que arriba la confirmació (només pot enviar una PDU sense confirmar).

Una conseqüència del protocol de finestra és que en el buffer de transmissió hi haurà com a màxim W PDUs emmagatzemades, on W és la mida de la finestra. Això permet dimensionar la mida del buffer de transmissió i de recepció, ja que hauran d’emmagatzemar com a màxim W PDUs.



3.1.4.2 Finestra òptima:

Si la mida de la finestra és massa petita, pot passar que el protocol sigui ineficient pel temps d’espera de les confirmacions. La figura 3.15 mostra un exemple on la finestra val W = 3. Després de transmetre tres PDUs el primari es queda bloquejat fins que arriben les confirmacions i es torna a repetir el cicle. Si el temps que està enviant les PDUs val TU i el cicle es repeteix cada TC, l’eficiència del protocol valdrà E = TU/TC.



Si la finestra del protocol és massa gran, també pot ser un inconvenient. Per exemple, perquè, com hem vist abans, els buffers de transmissió i recepció han de dimensionar-se per poder emmagatzemar un nombre de PDUs igual a la finestra.

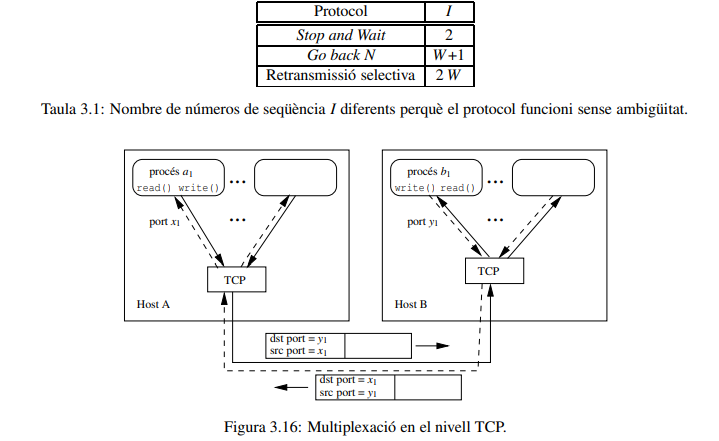
Per això es defineix la finestra òptima com la mínima finestra que permet assolir la velocitat efectiva màxima. És a dir, si fem servir una finestra inferior a l’òptima, vol dir que la velocitat efectiva serà inferior a la que podríem aconseguir amb una finestra més gran. En canvi, si augmentem la mida de la finestra més enllà de la finestra òptima, no augmentarem la velocitat efectiva més enllà de l’aconseguida amb la finestra òptima.

En l’exemple anterior, si la mida de les PDU es fixa amb un temps de transmissió de tt segons, aleshores la finestra òptima seria de W = [TU /tt].

3.1.4.3 Dimensionat del camp “número de seqüència” de les PDUs:

Tal com hem vist anteriorment, els protocols ARQ necessiten un número de seqüència per relacionar les PDU d’informació i les seves corresponents confirmacions. El número de seqüència el porta un camp de la capçalera de la PDU. Si aquest camp té n bits, aleshores el número de seqüència podrà tenir un valor en l’interval [0...2n −1]. És a dir, tenim un nombre de números de seqüència diferents de I = 2n . Quan s’assoleix el número de seqüència 2n −1, es torna a començar amb el valor 0 i es repeteix el “cicle” de números de seqüència.

La reutilització del números de seqüència pot crear ambigüitats si n no és prou gran. És a dir, quan es confirma la PDU k, com pot saber el primari si es confirma la PDU de l’actual “cicle”, o del “cicle” anterior? Si les PDUs poden arribar desordenades i amb retards arbitraris, aquesta ambigüitat no es pot resoldre. L’únic que podem fer és agafar un nombre de bits n prou gran perquè la probabilitat que es doni sigui gairebé zero (per exemple, això és el que fa TCP). En canvi, si les PDUs arriben en el mateix ordre que s’han enviat, es pot demostrar amb un nombre de números de seqüència I igual o major que el de la taula 3.1, el protocol funciona sense ambigüitats. Així, per exemple, amb stop and wait bastaria un sol bit pel camp amb el número de seqüència. En canvi, amb Go back N, si desitgem una finestra de mida W, necessitarem un nombre de bits n que compleixi 2n ≥ W +1 i amb Retransmissió selectiva 2n ≥ 2W.



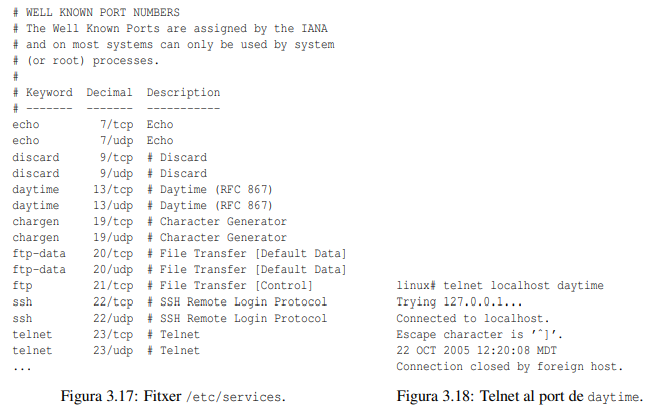
**3.2 El nivell de transport d’Internet**

El nivell de transport proveeix un canal lògic de comunicació entre les aplicacions. Per exemple, en la pràctica el nivell d’aplicació està format per dos processos que es comuniquen entre ells fent servir la xarxa, ja que el nivell de transport permet que els processos es comuniquin entre ells com si estiguessin en el mateix computador, tot i que possiblement estiguin en computadors situats en punts geogràficament distants. Per aconseguir-ho, el nivell de transport implementa un protocol punt-a-punt entre els dos hosts que es comuniquen que fa un “multiplexat” de la informació transmesa pels processos.

A Internet hi ha dos protocols de nivell de transport: UDP i TCP. El multiplexat s’aconsegueix amb un identificador de 16 bits anomenat “port” que identifica els processos que es comuniquen. La figura 3.16 il·lustra aquest funcionament.

En TCP/IP les connexions solen seguir el “paradigma client-servidor”. En aquest model el “servidor” (normalment un daemon en una màquina UNIX) espera les peticions dels “clients”. Per això, el servidor “escolta” les peticions adreçades a un port well-known. Els ports well-known tenen un valor en l’interval [0, ..., 1023] i estan assignats per IANA. El client és sempre el que inicia la connexió cap al servidor i té un port assignat pel sistema operatiu en l’interval [1024, ..., 2 16 −1]. Aquest port s’anomena “efímer” perquè només identifica el procés del client mentre dura la connexió.

En una màquina UNIX el fitxer /etc/services té el llistat dels ports well-known (vegeu la figura 3.2). En aquest fitxer hi ha un nom que identifica el servei associat a cada port i que es pot fer servir en algunes aplicacions per identificar el port. Per exemple, l’aplicació Telnet es connecta per defecte al port 23 (sessió Telnet). Aquesta comanda, però, admet un segon argument per indicar port diferent. En aquest cas, Telnet fa un bolcat per stdout de tot el que rep del servidor i envia al servidor tot el que llegeix de stdin. Per exemple, en la figura 3.2 hi ha el resultat de fer Telnet al port de daytime. El servidor d’aquest port envia el dia i l’hora i tanca la connexió.



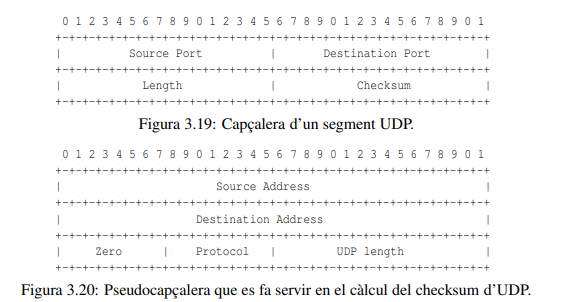
**3.3 El protocol UDP**

UDP és un protocol de nivell de transport “orientat al datagrama”. És a dir, ofereix el mateix servei que el transport de datagrames del nivell IP. Bàsicament, l’únic que fa és afegir la capçalera de la figura 3.19 a la informació que rep de l’aplicació per construir un “datagrama UDP” i passar-lo al nivell IP per a la seva transmissió. Així doncs, el servei UPD és **no orientat a la connexió (connectionless)** i **no fiable (si el datagrama UDP es perd, UDP no el retransmet)**. Una altra característica importat és que cada operació d’escriptura del nivell d’aplicació genera un datagrama UDP.

Les aplicacions en “temps real” fan servir UDP. Exemples d’aquest tipus d’aplicacions són telefonia o videoconferència sobre TCP/IP. Aquestes aplicacions no toleren retards molt variables. Això és perquè l’aplicació en recepció llegeix un buffer a un ritme constant. Si un datagrama arriba més tard de l’instant en què tocaria llegir-se, aleshores ha de descartar-se perquè és inservible per a l’aplicació. Per un altre costat, l’aplicació tolera un cert percentatge d’error: això es tradueix en un soroll en el so o imatge, que no impedeix la comunicació.

La **capçalera UDP** (figura 3.19) té una mida fixa de 8 bytes. Hi ha 4 camps:

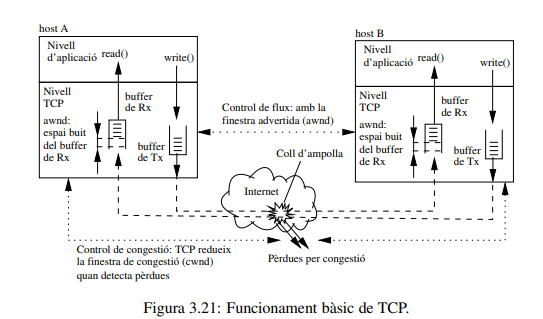
* Port font i destinació, per identificar el processos que es comuniquen.
* Length amb la mida total del datagrama UDP (payload UDP + 8).
* Checksum: protegeix la capçalera i el camp de dades. Es calcula aplicant l’algorisme de checksum conjuntament a una “pseudo capçalera”, la capçalera UDP i el camp de dades. La “pseudo capçalera” té alguns camps de la capçalera IP per fer una doble comprovació que el datagrama ha arribat a la destinació correcta (vegeu la figura 3.20). En UPD el checksum és opcional (tot i que sol fer-se servir). En cas de no fer-se servir s’envia un checksum igual a 0. A causa d’aquesta pseudo capçalera, si es fa servir NAT, cal recalcular el checksum de la capçalera UDP.



**3.4 El protocol TCP**

TCP és el protocol de nivell de transport que es fa servir en Internet per a la transmissió fiable d’informació. TCP és un protocol extrem a extrem, ARQ, orientat a la connexió, del que podríem destacar els següents objectius:

* Recuperació d’errors, per tenir una transmissió fiable.
* Control de flux, perquè el primari no enviï els segments a més velocitat de la que pot processar-los el secundari. Altrament, hi hauria segments que es podrien perdre per vessament del buffer de recepció del secundari.
* Control de congestió, perquè el primari no enviï els segments a més velocitat de la que pot processar-los la xarxa. Altrament hi hauria segments que es podrien perdre en el “coll d’ampolla” de la xarxa. El coll d’ampolla és l’enllaç en què els segments que envia el primari veuen una velocitat mitjana més petita. Si el primari envia segments a una velocitat major, s’omplirà el buffer del router que ha d’enviar els segments per aquest enllaç, i es produiran pèrdues. En les connexions que travessen Internet això passa amb relativa freqüència. Les pèrdues per congestió no són un mal funcionament de TCP, a diferència del que serien les pèrdues degudes a un control de flux. Això és perquè les pèrdues per control de flux es poden evitar fàcilment (tal com es veurà més endavant). Les pèrdues per congestió, en canvi, serveixen de senyal a TCP per adonar-se que ha superat la velocitat que imposa el coll d’ampolla.
* Segments de mida òptima: a diferència d’UDP, TCP va agafant bytes de l’aplicació per generar segments de mida òptima. TCP manté una variable anomenada Maximum Segment Size (MSS), que és la mida del que considera òptima pel payload (camp de dades). Típicament, la mida òptima és la major possible (per minimitzar l’overhead de les capçaleres), però que no produeixi fragmentació a nivell IP.



La figura 3.21 mostra el funcionament bàsic de TCP. En cada extrem, TCP manté un buffer de transmissió i un de recepció (buffer de Tx i Rx en la figura). L’aplicació fa servir les crides al sistema operatiu read() i write() per llegir i escriure en aquests buffers. Quan el buffer de Tx està ple, la crida write() queda bloquejada. Quan TCP ha enviat dades del buffer de Tx i s’han confirmat, TCP les esborra del buffer i queda espai lliure perquè l’aplicació hi escrigui. Anàlogament, la crida read() queda bloquejada fins que arriben noves dades. Cada vegada que arriba un segment, TCP envia una confirmació (hi ha algunes excepcions, per exemple, si es fa servir la tècnica dels delayed acknowledgments).

**Control de flux**: Per evitar el vessament del buffer de Rx (en cas que el primari enviï a una velocitat major de la que el secundari processa les dades), TCP fa servir la “finestra advertida”(advertised window). La finestra advertida és una de les informacions que sempre s’envien en la capçalera TCP. Quan TCP rep un segment, guarda el valor de la finestra advertida en la variable awnd. El primari no pot enviar mai més bytes sense confirmar dels que diu awnd. D’aquesta manera, sempre hi haurà espai suficient per guardar els bytes enviats en el buffer d’Rx. Si el buffer d’Rx s’omple, el secundari enviarà una finestra advertida igual a 0 i el primari es quedarà bloquejat fins que el secundari torni a enviar una finestra advertida major que 0.

Cal notar que aquest mecanisme de control de flux és diferent a l’explicat en els algorismes ARQs bàsics: en els ARQs bàsics hem suposat que la confirmació s’envia quan el nivell superior llegeix la informació del buffer de recepció del secundari. D’aquesta manera, si el secundari “buida” el buffer de recepció més lentament de com arriben les PDUs, l’enviament de les confirmacions es retarda i el primari redueix conseqüentment la tassa de transmissió de les PDUs. TCP, en canvi, envia la confirmació immediatament després de rebre el segment de dades. El control de flux s’aconsegueix amb la finestra advertida: si l’aplicació llegeix més lentament el buffer de recepció de com arriben els segments d’informació, el buffer s’omple i la finestra advertida es redueix. Això fa que el primari redueixi conseqüentment la tassa de transmissió dels segments d’informació.

**Control de congestió**: quan es produeixen pèrdues en algun coll d’ampolla de la xarxa, vol dir que el primari ha de reduir el nombre de segments que envia, és a dir, ha de reduir la finestra de transmissió. Per a fer aquest control, TCP fa servir una altra finestra: La “finestra de congestió” (congestion window). TCP manté la variable cwnd amb el valor d’aquesta finestra. Aquesta variable s’incrementa si no es detecten pèrdues i es decrementa en cas contrari. En la secció 3.4.5 s’explicaran els algorismes bàsics que regulen cwnd. Per adaptar-se a la condició més restrictiva (control de flux o congestió), TCP fa servir en tot moment una finestra de transmissió (wnd) igual a: wnd = min{awnd,cwnd}

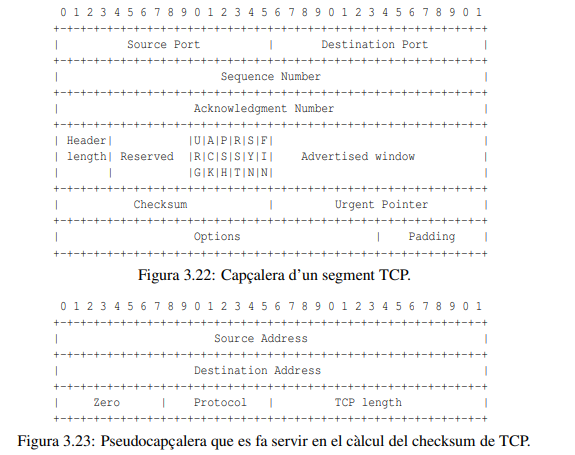
Cal destacar que TCP no sempre ha de limitar la velocitat de transmissió de la connexió. De fet, podem classificar les connexions en:

* Bulk transfer (massives): durant la transmissió l’aplicació sempre té dades llestes per enviar. En aquest cas, el buffer de transmissió sempre està ple i TCP envia segments de mida màxima. La finestra de transmissió limita la velocitat efectiva de la connexió. Exemples d’aplicacions que generen aquest tipus de tràfic són ftp, web, mail, etc.
* Interactives: són aplicacions en què l’usuari interactua amb la màquina remota, com ara amb l’aplicació Telnet. En aquest cas, la informació s’envia en missatges de pocs bytes i de forma discontínua. TCP no necessita introduir limitacions en aquest tipus de transmissions a causa del poc tràfic que generen, és a dir, la quantitat de dades que s’envien està molt per sota del que permet la finestra de transmissió.

Tot i això, aquestes aplicacions poden representar un problema. Considerem per exemple l’aplicació Telnet. L’usuari possiblement estarà bastants milisegons entre tecla i tecla, més que el temps que passa des que s’envia un segment fins que arriba la corresponent confirmació (ack). D’aquesta manera, cada vegada que arribés un ack, TCP es trobaria que en el buffer de transmissió hi hauria un sol byte! El resultat seria que es generaria un segment i la seva corresponent confirmació per cada tecla pitjada. La transmissió de molts paquets tan petits en Internet reduiria l’eficiència. Per solventar aquest problema TCP, opcionalment, pot fer servir dos mecanismes:

* + Delayed acknowledgments: consisteix en que el secundari no transmet l’ack immediatament després de rebre noves dades. En canvi, espera un temps per veure si arriben nous segments d’informació i després envia un ack (que els confirmarà a tots). L’RFC de Host requirements diu que el retard dels delayed acknowledgments ha de ser sempre inferior a 0,5 segons i que ha d’enviar-se almenys 1 ack per cada dos segments de mida màxima. Els retards dels delayed acknowledgments de les implementacions de TCP/IP que hi ha en la pràctica són com a màxim de 200 ms.
  + Nagle algorithm: l’algorisme anterior redueix el nombre d’acks, però no evita el problema “d’un segment per cada tecla pitjada” de les aplicacions interactives. Per resoldre aquest problema en l’RFC [23] es proposa el següent algorisme: cada vegada que arriben noves dades al buffer de transmissió de TCP i la finestra permet enviar un nou segment, només s’envia si: (i) hi ha prou bytes per enviar un segment de mida màxima o (ii) no hi ha bytes pendents de confirmar. Si hi ha bytes pendents de confirmar, els bytes que van arribant al buffer de transmissió es retenen fins que arriba la confirmació. El fet de retardar l’enviament de nous segments fins que arriba la confirmació dels anteriors afavoreix que en una connexió interactiva el primari tingui més bytes disponibles per enviar quan arribi la confirmació.

**3.4.1 Capçalera TCP**



La figura 3.22 mostra el format de la capçalera TCP. El significat dels camps és el següent:

* Source Port i Destination Port: Port font i destinació.
* Sequence Number: número de seqüència del segment (s’explica en la secció 3.4.2).
* Acknowledgment Number: confirmació (s’explica en la secció 3.4.2).
* Header length: mida de la capçalera en words de 32 bits (igual que el camp header legnth de la capçalera IP). Igual que en la capçalera IP, la mida mínima de la capçalera TCP és de 20 bytes (i el camp header legnth té el valor 5). La mida màxima de la capçalera TCP és de 15·4 = 60 bytes.
* Reserved: bits reservats per possibles ampliacions del protocol. Es posen a 0.
* Flags: són els següents:
  + URG: (Urgent) Indica que es fa servir el camp Urgent Pointer explicat més avall.
  + ACK: (Acknowledgment) Indica que es fa servir el camp Acknowledgment.
  + PSH: (Push) Indica que s’ha de deixar llegir el buffer de Rx del secundari el més aviat possible. L’activació d’aquest flag depèn de la implementació. Les implementacions derivades de BSD l’activen quan el buffer de Tx es queda buit.
  + RST: (Reset) S’activa quan es vol avortar la connexió. Un exemple és quan es rep un segment d’un client adreçat a un port on no hi ha cap servidor escoltant. En aquest cas, TCP contesta amb un segment amb el flag de reset activat.
  + SYN: (Synchronize) Es fa servir en l’establiment de la connexió (s’explica en la secció 3.4.3).
  + FIN: (Finalize) Es fa servir en la finalització de la connexió (s’explica en la secció 3.4.3).
  + Advertised window: Indica la finestra advertida.
* Checksum: A diferència d’UDP, a TCP el càlcul del checksum és obligatori. Per al càlcul del checksum s’agafa:
  + La pseudo capçalera que mostra la figura 3.23. Aquesta pseudo capçalera té alguns camps de la capçalera IP (adreça font, destinació i protocol) i la mida del segment TCP (capçalera més payload). Tot i que la mida del segment TCP no es posa en la capçalera TCP, es té en compte en el càlcul del checksum.
  + La capçalera TCP.
  + El camp de dades (payload) del segment.

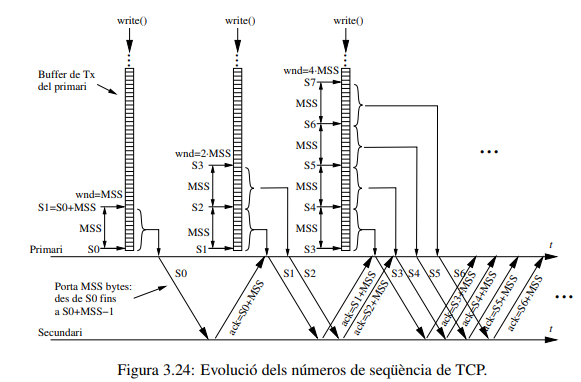
Igual que en UDP, els camps de la capçalera IP de la pseudo capçalera s’inclouen per tenir una major certesa que el segment no ha arribat a una destinació equivocada. A causa de la pseudo capçalera, si es fa servir NAT cal recalcular el checksum de la capçalera TCP.

* Urgent Pointer: Implementa un mecanisme per indicar dades “urgents” (és a dir, que s’han d’atendre el més aviat possible). Les dades urgents anirien del primer byte del segment fins al byte indicat per l’Urgent Pointer. Aquest flag es fa servir rares vegades. Un exemple és quan tecleja un control-C (interrupció) des de l’aplicació Telnet.
* Options: De forma anàloga al protocol IP, TCP permet afegir opcions a la capçalera. A diferència d’IP, però, les opcions de TCP solen fer-se servir. L’RFC de TCP [21] dona el format de les opcions i en descriu algunes. D’altres s’han proposat en RFCs posteriors. Les opcions que més es fan servir són:
  + Maximum segment size[21]: Es fa servir durant l’establiment de la connexió per suggerir el valor del MSS a l’altre extrem (vegeu la secció 3.4). El valor que la suggereix és la MTU de la xarxa on està connectada d’interfície menys la mida de la capçalera IP i TCP (sense opcions). Per exemple, si la xarxa és una ethernet (MTU 1500), aleshores es posa un valor de 1460.
  + Window scale factor[32]: Es fa servir durant l’establiment de la connexió per indicar que el valor de la finestra advertida s’ha de multiplicar per aquest factor d’escala. Això permet advertir finestres majors de 216 bytes.
  + Timestamp [32]: Es fa servir en el càlcul de l’RTT (vegeu la secció 3.4.6).
  + SACK [43]: Permet que TCP faci retransmissió selectiva (selective ack). TCP fa servir el camp ack per indicar fins on s’ha rebut correctament (vegeu la secció 3.4.2). Amb l’opció SACK el secundari pot indicar blocs de segments que s’han rebut correctament més enllà del segment confirmat per l’ack. D’aquesta manera, el primari pot triar millor els segments que s’han de retransmetre.
* Padding: Bytes de “farciment” afegits perquè la capçalera tingui un múltiple de 32 bits.

**3.4.2 Números de seqüència en TCP**

En TCP els números de seqüència i les finestres es mesuren en bytes. És a dir, si un segment porta S bytes, el número de seqüència del pròxim segment s’incrementarà en S (i no 1 com fèiem en els protocols ARQ explicats en la secció 3.1). La figura 3.24 mostra l’evolució dels números de seqüència a partir del primer segment de dades d’una connexió TCP. El número de seqüència que porta la capçalera (*Si* en la figura) identifica el primer byte de dades del segment. Inicialment la finestra val MSS (només permet enviar un segment) i s’incrementa amb MSS cada vegada que arriba un ack que confirma noves dades (això s’explica en detall en la secció 3.4.5).

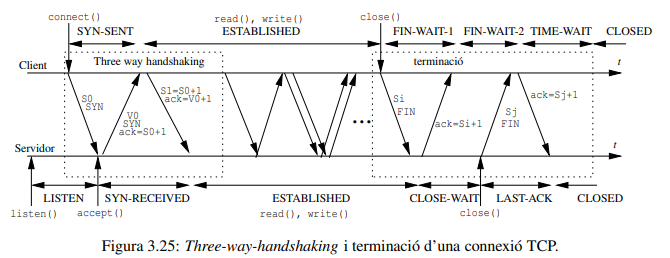
Les confirmacions porten un valor igual al número de seqüència que porta el segment que confirmen, més el nombre de bytes de dades del segment. És a dir, si el segment amb número de seqüència *Si* porta MSS bytes (segment de mida màxima), aleshores la confirmació porta el valor: ack = *Si* + MSS (vegeu la figura 3.24). El valor que porta l’ack és, doncs, igual al número de seqüència del pròxim segment de dades que espera rebre el secundari. Amb altres paraules, és el pròxim byte que falta al secundari. Com que les confirmacions són acumulatives, quan el primari rep la confirmació interpreta que tots els bytes identificats amb números de seqüència inferiors al de l’ack han arribat correctament al secundari i els esborra del buffer de transmissió.

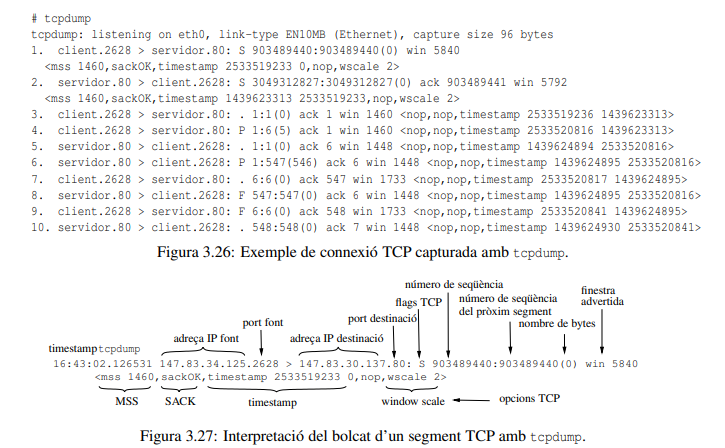


**3.4.3 Establiment i terminació d’una connexió**

TCP La figura 3.25 mostra las fases d’establiment i terminació d’una connexió TCP. La figura mostra també els estats per què passa el client i el servidor durant la connexió i les crides al sistema. Els estats s’expliquen en més detalls en la pròxima secció, on s’explica el diagrama d’estats de TCP. Per a més informació sobre les crides al sistema, en l’apèndix 3.A hi ha un exemple de la programació d’un client i un servidor TCP.

L’establiment d’una connexió TCP es coneix amb el nom de three-way-handshaking i consisteix sempre en l’intercanvi de tres segments que no porten dades (només la capçalera TCP): SYN/SYN+ack/ack (vegeu la figura 3.25). El significat d’aquests paquets és el següent:

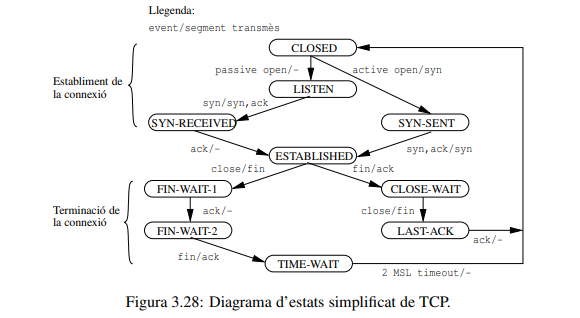
* El primer segment (SYN) sempre l’envia el client. Aquest segment té típicament un port well-known (< 1024) com a port destinació i un port efímer (≥ 1024) com a port origen. La característica més important és que té el flag de SYN activat. Aquest segment és un dels pocs casos en què, per motius obvis, no es confirma res (el flag d’ACK ha d’estar desactivat). Aquest segment típicament porta opcions com ara l’MSS o les opcions timestamp, window scale factor o SACK (vegeu la secció 3.4.1), com a indicació que es volen fer servir. Aquest segment també porta l’initial sequence number, que és el número de seqüència inicial que es farà servir per identificar els bytes de dades enviats pel client. Aquest número és un número aleatori de 32 bits.
* El segon segment (SYN+ack) l’envia el servidor. També té el flag de SYN activat i porta l’initial sequence number per identificar els bytes de dades enviats pel servidor. Els segments de SYN, tot i no portar cap byte de dades, consumeixen un número de seqüència. L’ack d’aquest segment, per tant, té un valor igual a l’initial sequence number del client més 1.
* Finalment, el client confirma la recepció del SYN+ack enviant la confirmació amb l’initial sequence number del servidor més 1.

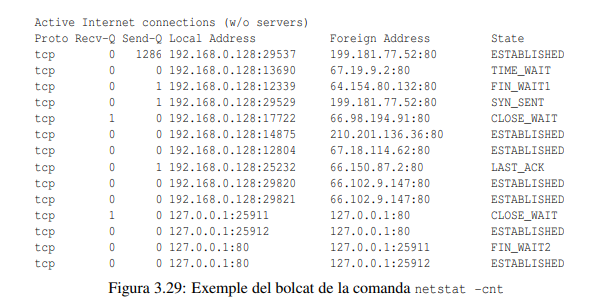


La terminació de la connexió es produeix quan es produeix l’intercanvi de 3 o 4 segments: FIN/ack en un sentit i FIN/ack en el sentit contrari (vegeu la figura 3.25). El segment de FIN s’envia quan l’aplicació fa la crida close(). Igual que el SYN, el FIN consumeix un número de seqüència. Una diferència important és que el segment de FIN pot portar dades. De fet, és normal que això passi si encara hi ha bytes de dades per enviar en el buffer de transmissió de TCP quan l’aplicació fa la crida close(). Un altre detall és que fins que l’aplicació no fa la crida close(), pot continuar escrivint i TCP enviarà les dades. Si l’altre extrem ja ha executat la crida close(), aleshores es diu que la connexió està en estat half closed, i només es poden enviar dades en un sentit (el que encara no ha enviat el segment de FIN). A diferència del three-way-handshaking, on el client envia sempre el primer segment de SYN, el primer segment de FIN pot enviar-lo el client o el servidor.

La figura 3.26 mostra una connexió TCP capturada amb la comanda tcpdump. La connexió consisteix en la descàrrega d’una pàgina html d’un servidor web (port 80). Per a la interpretació del bolcat, la figura 3.27 mostra el significat del bolcat d’un segment TCP de SYN capturat amb tcpdump. Cal destacar que després de capturar el segment de SYN, tcpdump resta l’initial sequence number als números de seqüència perquè siguin més fàcils de llegir.

Per reduir la mida de les línies del bolcat de la figura 3.26 s’ha substituït el timestamp de tcpdump per un número de línia (d’1 a 10). En la figura es pot comprovar com els segments de SYN i FIN consumeixen un número de seqüència. En la descàrrega de la pàgina html el servidor envia un únic segment TCP d’informació (amb 546 bytes, vegeu la línia 6). En aquest exemple es pot comprovar com els segments del three-way-handshaking no porten bytes d’informació. En aquest cas, a més, s’envien només 3 segments en la terminació de la connexió, perquè l’ack del primer segment de FIN s’envia conjuntament amb el segon segment de FIN.





**3.4.4 Diagrama d’estats de TCP**

La figura 3.28 mostra el diagrama d’estats simplificat que especifica el RFC de TCP [18]. En la part superior hi ha els estats pels que passa TCP durant l’establiment de la connexió i en la part inferior els que passa durant la terminació.

En la fase d’establiment el servidor fa un passive open i passa a l’estat de LISTEN a l’espera de peticions dels clients. Quan s’executa la crida connect() en el client, aquest envia un segment de SYN (vegeu la figura 3.25). Quan el servidor rep aquest segment, si accepta la connexió (fa la crida accept(), aleshores el servidor passa a l’estat SYN-RECEIVED i envia un segment amb els flags de SYN i ACK activats. Quan el client rep el segment SYN, ack passa a l’estat ESTABLISHED i envia un ack. A la vegada, quan el rep el servidor també passa a l’estat de ESTABLISHED i la connexió queda establerta.

L’explicació de la terminació de la connexió és anàleg (compareu les figures 3.25 i 3.28). Ara, però, el host que ha iniciat la terminació (que ha enviat el primer segment de FIN), quan rep el segment de FIN de l’altre extrem envia l’ack i es queda en l’estat de TIME-WAIT durant un temps igual a 2 vegades l’anomenat Maximum Segment Lifetime (MSL). Aquest és el temps que se suposa que pot estar un datagrama com a màxim a Internet, i sol´ agafar-se igual a 1 minut (és a dir, està en l’estat TIME-WAIT durant 2 minuts). El motiu és que la confirmació que ha enviat es podria perdre i es podrien rebre retransmissions de l’últim FIN. El host contestaria aquestes possibles retransmissions mentre està en aquest estat. L’altre extrem, en canvi, quan rep l’ack del FIN que ha enviat té la certesa que els dos extrems han rebut el segment de FIN i passa directament a l’estat de CLOSED.

En una màquina UNIX es poden veure les connexions TCP que hi ha establertes i el seu estat amb la comanda netstat -t. La figura 3.29 mostra un exemple obtingut en una màquina Linux després d’haver accedit a diferents servidors web (port 80) amb un navegador. Les opcions de la comanda netstat indiquen: Que refresqui contínuament el bolcat de les connexions (-c), que mostri les adreces en format numèric ( -n) i que només mostri les connexions TCP (-t). Les columnes Recv-Q i Send-Q indiquen, respectivament, els bytes que hi ha en els buffers de recepció i transmissió de TCP (els bytes que encara no ha llegit l’aplicació i que encara no ha confirmat el secundari). Fixeu-vos que s’ha fet servir NAT (l’adreça del localhost és privada) i que hi ha dues connexions a un servidor web instal·lat en el mateix host (últimes 4 entrades del bolcat).

**3.4.5 Control de congestió**

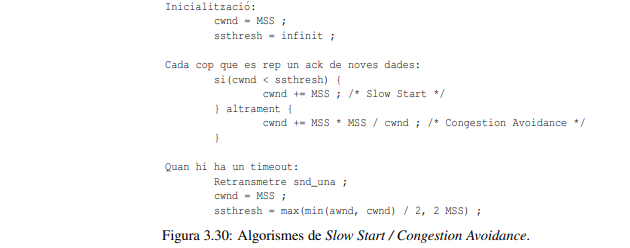
Tal com s’ha explicat a l’inici de la secció 3.4, una de les tasques fonamentals de TCP és el control de congestió. És a dir, adaptar-se a la velocitat de transmissió que fixa el coll d’ampolla de la xarxa. Sense control de congestió, Internet es tornaria inestable i es col·lapsaria. El control de congestió consisteix en una “autoregulació” que fa TCP mitjançant la finestra anomenada Congestion window (cwnd). La regulació de cwnd és del tipus multiplicative decrease, additive increase. Això vol dir que quan TCP detecta congestió, redueix el valor de cwnd de forma multiplicativa (és a dir, ràpidament) per evitar agreujar la congestió. En canvi, si no hi ha congestió, cwnd s’incrementa lentament i intenta buscar el punt on no hi hagi congestió, però amb màxim aprofitament de les línies de transmissió. En concret, TCP fa servir dues parelles d’algorismes bàsics per ajustar cwnd, i que s’expliquen a continuació:

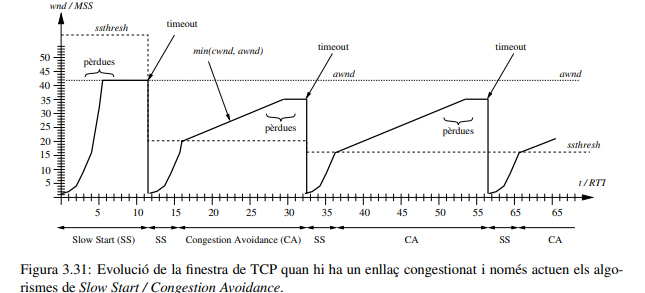
* Slow Start / Congestion Avoidance.
* Fast Retransmit / Fast Recovery.

3.4.5.1 Algorismes Slow Start / Congestion Avoidance

Els algorismes de Slow Start / Congestion Avoidance fan el control bàsic de la finestra de TCP. Aquest algorismes fan servir les següents variables:

* cnwd: és la finestra de congestió.
* Snd\_una: (unacknowledged) és el primer segment no confirmat. És a dir, és el segment que fa més temps que espera en el buffer de transmissió per ser confirmat.
* ssthresh: (slow start threshold) llindar entre les fases de slow start i congestion avoidance.



****

La figura 3.30 sumaritza el funcionament d’aquests dos algorismes. Quan s’estableix la connexió TCP, s’inicialitza cwnd = MSS: és a dir, TCP pot enviar només un segment sense confirmar (MSS bytes), d’aquí el nom de slow start. Mentre no hi ha pèrdues, TCP incrementa cwnd en MSS per cada nou ack: és a dir, quan arriba l’ack del primer segment, cwnd s’augmenta a 2 MSS i TCP envia 2 segments. Quan es reben els 2 acks, cwnd s’augmenta a 4 MSS i TCP envia 4 segments. Després seran 8, 16, 32, ...

Si els retards són grans i les línies de transmissió ràpides, els acks solen arribar en una ràfega després d’un Round Trip Trime (RTT). El RTT és el retard que hi ha des que TCP envia un segment fins que arriba la confirmació. Aquesta situació es dona típicament quan el host està connectat a una LAN i la línia d’accés a Internet és ràpida (per exemple, ADSL). En aquest cas, durant la fase de slow start cada RTT la cwnd es multiplica per 2, és a dir, en un temps n RTT, la cwnd valdrà aproximadament 2 n (augment exponencial). El nom d’slow start és contradictori des d’aquest punt de vista, perquè cwnd augmenta ràpidament durant el slow start.

Si es produeix alguna pèrdua, saltarà el TO, es farà la retransmissió i es posarà ssthresh al valor de la finestra de transmissió que hi havia en el moment de la pèrdua dividit per 2 (min(awnd, cwnd) / 2). Després es començarà amb slow start i cwnd augmentarà ràpidament fins al valor de ssthresh. A partir d’aquest moment, TCP entra en la fase de Congestion Avoidance i la finestra augmenta lentament: aproximadament 1 MSS cada vegada que es rebin les confirmacions de tota una finestra. La finestra cwnd permet enviar cwnd/MSS segments de mida màxima sense confirmar. Quan es reben les seves respectives confirmacions, cwnd haurà augmentat MSS \* MSS/cwnd \* cwnd/MSS = 1 MSS. La figura 3.31 mostra l’evolució típica en dent de serra de la finestra de TCP en aquestes condicions.

Cal destacar que si no hi ha un enllaç congestionat (com passa típicament quan el client i el servidor són dintre d’una mateix LAN), TCP està sempre en slow start. En aquest cas la finestra augmenta fins a la finestra advertida (awnd) i a partir d’aquest moment actua només el control de flux: la finestra de TCP és sempre igual a l’advertida.

La justificació d’aquest algorisme és la següent: la variable ssthresh intenta ajustar-se a un valor on TCP no té pèrdues. Per això slow start augmenta ràpidament el valor de la finestra fins aquest punt. A partir de l’ssthresh, la fase de Congestion Avoidance augmenta lentament cwnd per si la finestra ha quedat per sota del valor òptim i l’enllaç s’està infrautilitzant.

3.4.5.2 Algorismes Fast Retransmit / Fast Recovery

Per millorar l’eficiència, les implementacions actuals de TCP acompanyen els algorismes d’Slow Start / Congestion Avoidance com a mínim dels algorismes de Fast Retransmit / Fast Recovery.

Per identificar algunes de les diferents versions de TCP que s’han fet al llarg del temps s’han utilitzat noms de ciutats: Tahoe, Vegas, Reno. A la versió que implementa només Slow Start / Congestion Avoidance Fast Retransmit / Fast Recovery se la coneix amb el nom de TCP Reno. En la actualitat les implementacions milloren encara més el comportament de TCP amb algorismes addicionals: NewReno, SACK... que no explicarem.

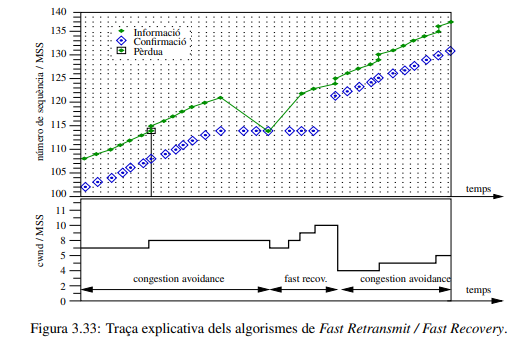
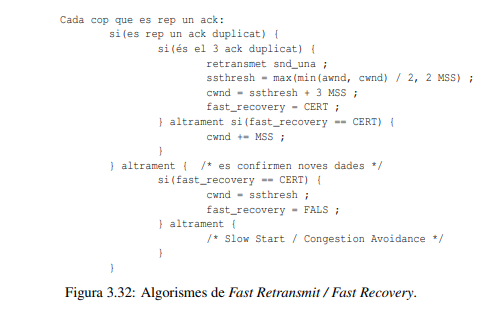
Quan el secundari de TCP rep un segment d’informació fora de seqüència, sempre envia una confirmació del segment que espera rebre (a diferència dels algorismes ARQ bàsics explicats en la secció 3.1, on suposàvem que el secundari deixava d’enviar acks). D’aquesta manera, quan es perd un segment, el secundari envia un ack amb el mateix número de seqüència cada vegada que rep un segment d’informació: el del segment perdut. Aquests acks s’anomenen acks duplicats.

Fast Retransmit / Fast Recovery es basen en l’observació que quan es reben confirmacions consecutives que no confirmen noves dades (confirmacions duplicades), són una indicació que s’han produït pèrdues. Per això, TCP retransmet el segment que sospita que s’ha perdut sense esperar que salti el timeout (això s’anomena fast retransmit). A més, intenta tornar a congestion avoidance sense haver de passar un altre cop per slow start (això s’anomena fast recovery).

La figura 3.32 mostra un pseudocodi explicatiu d’aquests algorismes. Quan TCP rep el tercer ack duplicat (és a dir, 4 acks amb el mateix número de seqüència):

* Retransmet el segment.
* Calcula el ssthresh com la meitat de la finestra actual (igual que quan salta el timeout).
* Fixa cwnd = ssthresh + 3 MSS: La finestra que suposa que hi havia abans de la pèrdua, més els 3 segments que han generat els acks duplicats.
* Passa a la fase de fast recovery. Durant la fase de fast recovery, per cada ack duplicat s’incrementa la finestra en un MSS, per poder enviar un nou segment (si ho deixa la finestra).
* Quan arriba un segment que confirma noves dades se surt de fast recovery i es posa cwnd = ssthresh, per iniciar la fase de congestion avoidance.

La figura 3.33 il·lustra el funcionament dels algorismes de Fast Retransmit / Fast Recovery. La part superior de la figura mostra una traça capturada en el primari amb l’evolució dels números de seqüència i les confirmacions, en un interval de temps on es produeix una pèrdua. La part inferior de la figura mostra l’evolució de la cwnd. Quan el primari rep 3 acks duplicats la finestra val cwnd = 8 MSS. En aquest moment el primari retransmet el segment perdut, calcula ssthresh = 4 MSS, fixa cwnd = 4 + 3 = 7 MSS i entra en la fase de fast recovery. En aquesta fase, el primari augmenta cwnd en MSS per cada ack duplicat. Com que quan entra en fast recovery hi ha 8 segments sense confirmar, el primari pot enviar nous segments després de rebre dos ack duplicats més (cwnd augmenta fins a 9 MSS). Quan es confirma el segment retransmès (primer ack que arriba després dels acks duplicats), el primari surt de fast recovery i posa cwnd = ssthresh = 4 MSS. Com que en aquest moment només hi ha 2 segments sense confirmar, el primari pot enviar-ne 2 més. A partir d’aquest moment, el primari entra en fase de congestion avoidance.



**3.4.6 Càlcul del RTO – RFC**

TCP manté un temporitzador per controlar les retransmissions: Retransmission TimeOut (RTO). Aquest temporitzador està activat sempre que hi ha dades pendents de confirmar. Cada vegada que arriba una confirmació de noves dades: (1) TCP calcula el valor de l’RTO. (2) Si hi ha més dades pendents de confirmar, aleshores el temporitzador s’actualitza al valor calculat, altrament el temporitzador es desactiva.

Per calcular l’RTO, TCP estima la mitjana (srtt) i desviació (rttvar) del Round Trip Time (RTT). Si la variable M té la mesura de l’RTT, aleshores l’RTO s’actualitza de la següent manera:

srtt = (1 - alpha) \* srtt + alpha \* M ;

rttvar = (1 - beta) \* rttvar + beta \* |srtt - M| ;

RTO = srtt + 4 \* rttvar ;

On alpha = 1/8 i beta = 1/4. Fixeu-vos que si M és constant, aleshores en successives iteracions srtt acabarà valent el valor d’M, i rttvar valdrà 0. El valor de l’RTO es fixa a la mitjana de l’RTT mesurat, més 4 vegades la desviació. És a dir, com més gran sigui la variabilitat del d’RTT, major serà la desviació i, per tant, la diferència entre el valor de l’RTO i l’estimació de l’RTT (per evitar que l’RTO salti prematurament i es facin retransmissions innecessàries).

Si salta l’RTO, es retransmet el segment snd una i es duplica el valor de l’RTO: RTO = RTO \* 2. D’aquesta manera si un mateix segment s’ha de tornar a retransmetre, TCP s’espera el doble del temps esperat en la transmissió anterior. Això es fa per evitar que la xarxa es pugui tornar inestable. Quan es tornen a confirmar noves dades (s’aconsegueix transmetre un segment amb èxit) aleshores l’RTO es torna a actualitzar segons les fórmules anteriors.

La figura 3.34 mostra un diagrama de temps explicatiu del temporitzador de retransmissió. En la figura el segment S4 es retransmet dues vegades. La figura mostra els instants en que s’actualitza el temporitzador amb una fletxa vertical. Aquest són els instants en que es confirmen noves dades, o es produeix una retransmissió. En cas de retransmissió, l’RTO es fixa a dues vegades el valor que tenia anteriorment.

Per mesurar l’RTT les implementacions actuals de TCP fan servir l’opció timestamp de la capçalera TCP. Aquesta opció consisteix a enviar un timestamp en la capçalera dels segments (el valor del rellotge del host quan TCP envia el segment) i afegir també un echo del timestamp del segment que confirmen. D’aquesta manera, quan arriba l’ack, la diferència entre el rellotge del host i el valor que porta l’echo del timestamp és una mesura acurada de l’RTT. En el bolcat de tcpdump de la figura 3.26 es pot veure el funcionament d’aquesta opció.

