MENU PRINCIPALE

Home
Servizi
Manuali
Software
AutocadGroup
Web link
Legislazione
Delphi
Advanced search
HTML editor

Seleziona lingua
Powered by Google Traduttore

Segui @compagotlc

Custom Sea Search

Donate

LOGIN

Nome utente

Password

Ricordami

Type the following code
29429

Secret

LOGIN

Password dimenticata?
Nome utente dimenticato?

ARTICOLI CORRELATI

Ripristino TCP e winsock su windows vista
Indirizzi Ip, Classi E Subnetting
Cosa è l'MTU
Guida al packet routing

Gestione avanzata del traffico internet

Domenica 09 Novembre 2008 17:34 amministratore

COSA SI INTENDE PER CONTROLLO DEL TRAFFICO IN UNA RETE E COME FUNZIONA

Per gestione del traffico si intende l'intero sistema di accodamento dei pacchetti in una rete in maniera tale da evitarne la congestione.

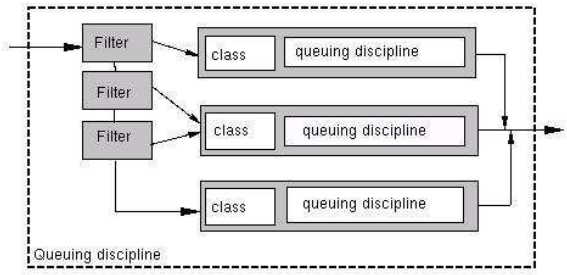
Il controllo del traffico è costituito da una serie di operazioni:

- **Classificazione** (classifying), con la quale i pacchetti vengono identificati e successivamente inseriti in un particolare flusso o classe.
- **Regolamentazione** (policing), con la quale viene limitato il numero di pacchetti o di bytes in un flusso che appartiene ad una particolare classe.
- **Programmazione** (scheduling), con la quale viene deciso in che modo i pacchetti sottoposti a controllo vengono ordinati e riordinati per la trasmissione.
- Lo **shaping,** che possiamo tradurre come modellazione, è il processo con il quale i pacchetti vengono ritardati e trasmessi in modo tale che il traffico sia regolare e rientri in una velocità programmata.

Tutte queste caratteristiche della gestione del traffico possono essere combinate insieme in modo da riservare la banda per un particolare tipo di traffico (o applicazione) o magari limitarne la banda.

Vediamo di analizzare ora la parte e di gestione della coda di uscita. Volendo comprendere più in profondità il controllo di traffico cominciamo ad introdurre i concetti chiave tramite i quali il kernel di Linux ragiona:

1. Queuing discipline
   A ogni dispositivo di rete può essere associata una disciplina di accodamento che effettua uno scheduling dei pacchetti in ingresso . Di default questa disciplina è una semplice FIFO.
2. Classes
   Le classi servono a definire le varie tipologie di traffico.Ogni classe e padrona di una coda la quale di default è una FIFO. Quando la disciplina di accodamento è chiamata questa applica i filtri per determinare la classe alla quale il pacchetto appartiene
3. Filters
   I filtri si occupano di smistare il traffico nelle diverse classi . Il concetto di filtro è molto più libero di quella di classe , infatti più filtri possono essere associati alla stessa classe e quindi alla stessa coda.



Sotto linux, il controllo del traffico è sempre stato una cosa complessa. Il comando da linea di comando **tc** ci fornisce una interfaccia per le strutture del kernel con le quali eseguire le nostre operazioni di shaping, scheduling, policing e classifying.
La sintassi di questo comando è comunque un po arcana. Il progetto *tcng* , ossia tc di nuova generazione, si propone di fornire una interfaccia un po più semplice, così da poter scrivere, capire e modificare in maniera più semplice e intuitiva i comandi che costituiscono la gestione del traffico.

Ora spiegato a grandi linee qual'è la struttura con la quale abbiamo a che fare andiamo a vedere i comandi con cui realizzarla e le funzionalità a cui possiamo attingere.

ELEMENTO	COMANDI
Queuing discipline	tc qdisc ...(OPTIONS)
Class	tc class ...(OPTIONS)
Filter	tc filter ... (OPTIONS)

La fonte più precisa di informazione sul traffic control sono proprio gli help in linea , usiamoli quindi come punto di partenza per capire i comandi e le possibili opzioni.

Per capire meglio i vari tipi di accodamento è necessario chiarire alcuni concetti e sopratutto la terminologia usata.

Queueing Discipline (**qdisc**) = Metodo di accodamento

E' un algoritmo che gestisce le code di un dispositivo, sia in arrivo(in ingresso) che partenza(in uscita).

root qdisc

La *root* qdisc è la qdisc collegata direttamente al dispositivo (qdisc radice).

Classless qdisc

E' una qdisc senza nessuna suddivisione interna, in pratica il metodo di accodamento non usa la suddivisione in classi dei dati.

Classful qdisc

Una classful qdisc usa più classi per selezionare i dati. Alcune di queste classi a loro volta potrebbero contenere altre qdisc, che posso essere a loro volta classful o classless. Seguendo questa definizione il metodo *pfifo_fast* è classfull, perché contiene al suo interno tre bande, le quali potrebbero essere interpretate come classi. Comunque, da una prospettiva di configurazione da parte dell'utente, dato che le sue classi non possono essere impostate col comando *tc*, questo metodo è considerato classless.

Classi

Una qdisc può avere molte classi. Ogni classe a sua volta può avere delle sotto classi. In questo modo una classe può avere come genitore una qdisc o un altra classe. In questo tipo di configurazione ad *albero* una classe foglia è una classe senza nessun'altra ramificazione. Questa classe ha una qdisc attaccata, la quale è responsabile dell'invio dei dati dalla classe. Quando viene creata una classe di default gli viene associata una qdisc fifo, mentre quando a quella stessa classe gli colleghi una o più classi figlie, queste rimpiazzeranno la qdisc fifo. Naturalmente è possibile rimpiazzare la qdisc fifo(classless) di default con una classfull in modo da potervi aggiungere altre classi.

Classificatore

Ogni qdisc di tipo classfull ha bisogno di capire verso quale classe inviare i dati, questa azione è svolta dal classificatore.

Filter

La classificazione può essere ottenuta usando i filtri. Un filtro contiene un certo numero di condizioni(regole) che, se vengono rispettate dal pacchetto in arrivo, consentono la sua trasmissione.

Scheduling

Una qdisc può, con l'aiuto di un classificatore, decidere che alcuni pacchetti abbiano la priorità su altri. questo processo è chiamato *Scheduling* o programmazione, ed è svolto ad esempio dalla qdisc di tipo *pfifo_fast*. Lo scheduling a volte è chiamato "riordinamento".

Shaping

Il processo di ritardo o di scarto dei pacchetti prima che vengano messi in uscita è chiamato shaping. Lo shaping avviene solo in uscita.

Policing

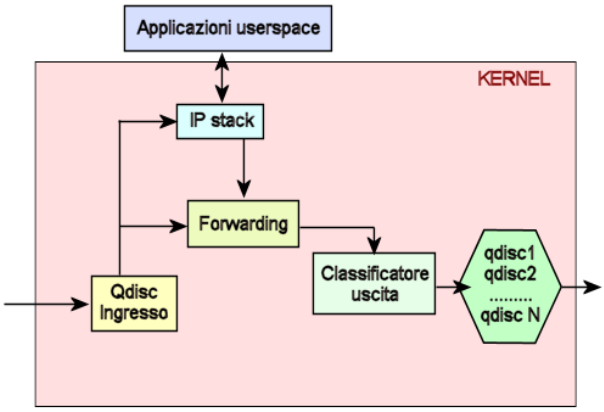
Dovrebbe Ritardare o scartare pacchetti affichè il traffico stia sotto una banda impostata, ma in linux con policing si intende solo lo scarto e non il ritardo dei pacchetti, infatti in ingresso non si ha nessuna coda, il pacchetto o passa o viene scartato.

Work-Conserving

Una qdisc *work-conserving* manda un pacchetto se ce ne è uno disponibile. In altre parole, questo tipo di metodo non ritarda mai un pacchetto se la scheda di rete è pronta a mandarlo (nel caso di una qdisc in uscita).

non-Work-Conserving

Alcuni tipi di accodamenti , come ad esempio il Token Bucket Filter, possono aver bisogno di trattenere un pacchetto per un certo periodo di tempo affinché vengano rispettati dei limiti sulla larghezza di banda(velocità). Questo significa che a volte potrebbe non inviare un pacchetto sebbene questo sia disponibile.



Il blocco più grande rappresenta il kernel. Il traffico entra da una interfaccia di rete e alimenta la Qdisc di ingresso la quale applica i Filtri ai pacchetti, decidendo così se scartarli o meno. Questa azione è chiamata "Policing".

Questo accade solo in questa prima fase, prima che il kernel li elabori ulteriormente. Scartando in questo punto i pacchetti indesiderati si risparmia un bel po della potenza di calcolo.

se al pacchetto è consentito di continuare potrebbe essere destinato ad una applicazione locale, nel qual caso entra nell' IP stack per essere processato e successivamente viene preso in carico dal programma (userspace area).
Il pacchetto potrebbe anche essere trasmesso senza passare dentro una applicazione, e in quel caso destinato verso l'uscita. Anche dalle applicazione nello Userspace possono essere inviati dei dati verso l'uscita.

Nel loro cammino verso l'uscita sono esaminati e instradati verso il classificatore in uscita.

Qui sono controllati, selezionati e accodati in un certo numero di qdisc. Se non ci fosse nulla configurato in uscita, allora di default troveremo solo una qdisc del tipo pfifo_fast. Questo processo è chiamato "queueing" ("accodamento").

Adesso il pacchetto sta dentro una qdisc, aspettando che il kernel richieda la sua trasmissione verso una interfaccia di rete. Questo processo è chiamato "dequeueing".

Per maggior chiarezza nel disegno l'ingresso e l'uscita sono disegnati in due posizioni distinte, ma questo non vuol dire che il sistema abbia 2 interfacce fisiche distinte.

I tipi di accodamento possono essere innanzitutto di due tipi:

- **Ingress qdisc** che si trovano all'ingresso di una interfaccia
- **Egress qdisc** che si trovano all'uscita di una interfaccia

La maggior parte dei successivi casi trattati tratterà di qdisc di tipi egress, lasciando alla fine la trattazione su quelle in ingresso, con i relativi esempi.

Metodi di accodamento per al gestione della banda

Dal kernel Linux 2.2/2.4 in poi la gestione della banda è comparabile ai sistemi professionali in commercio, anzi potremo dire che linux si spinge ben oltre quelle che sono le potenzialità della tecnologia Frame e ATM.

Giusto a scanso di equivoci, il nostro comando tc usa le seguenti regole per quanto riguarda il controllo della banda:

```
mbps = 1024 kbps = 1024 * 1024 bps (bps => byte/s)
mbit = 1024 kbit (kbit => kilo bit/s).
mb = 1024 kb = 1024 * 1024 b (b => byte)
mbit = 1024 kbit (kbit => kilo bit).
```

Internamente, il numero è memorizzato in bps (bit al secondo) e b (bit).
Ma quando tc stampa le velocità, usa la seguente sintassi:

```
1Mbit = 1024 Kbit = 1024 * 1024 bps => byte/s
```

Mostrando quindi i valori in byte, cioè ottetti di bit.

Tramite l'accodamento noi determiniamo in che modo i dati vengono trasmessi. Con il controllo del traffico quindi noi possiamo solamente modellare il flusso dei dati trasmessi, non modificarne il contenuto. Nel modo in cui internet funziona non abbiamo alcun controllo su cosa le altre persone ci inviano, un po come la buca delle lettere noi non abbiamo alcun modo di bloccare i dati a noi indirizzati se non contattando i vari mittenti.

Ad ogni modo Internet è per lo più basata sul protocollo TCP/IP che ha alcune caratteristiche che ci possono aiutare. Questo protocollo non ha nessun modo di conoscere la capacità della rete tra due host, così inizia a mandare i dati sempre più velocemente ("partenza lenta") e quando la connessione inizia a perdere pacchetti, perché non c'è più spazio per mandarli, allora incomincia a rallentare, in modo da permettere che tutti i pacchetti non debbano essere ritrasmessi e arrivino a destinazione.
Se voi aveste un router e voleste prevenire che certi host all'interno della vostra rete possano scaricare troppo velocemente, impedendo magari agli altri di lavorare, allora potreste utilizzare delle regole di shaping sulla interfaccia interna del router per quel determinato host che sta scaricando.

Potreste anche dovervi occupare del collo di bottiglia in un link, se per esempio aveste una scheda di rete da 100Mbit ed un router con una connessione ad internet da 256kbit, dove se non regolassimo noi stessi il flusso di dati che arriva al router, cioè se gli arrivassero più dati di quelli che lui è in grado di gestire, allora sarebbe lui stesso a ridurre la banda e quindi il flusso di dati, ma probabilmente in maniera diversa da quella che ci servirebbe. Quindi dovremo "*impossessarci dell'accodamento*" ed essere noi a regolare quali dati spedire o ricevere limitandone la banda.

Semplici metodi di accodamento "*senza classi*"

Come già detto con i metodi di accodamento non cambiamo il modo con cui i dati ci vengono mandati. Gli accodamenti "*senza classi*" sono quelli che a fronte di una grande quantità di dati in arrivo, alcuni di questi vengono o fatti passare o trattenuti (ritardati) oppure cestinati.

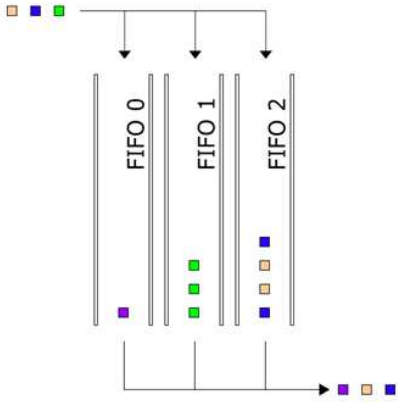
Questi metodi possono essere usati per modellare il traffico di una intera interfaccia, senza alcuna suddivisione.

Il più usato dei metodi senza classi è il *pfifo_fast qdisc* , che viene usato come default e non è altro che una classica coda *First In First Out*.
Tutti i vari metodi avanzati derivano da quest'ultimo, e ne rappresentano solo una variante.

- PFIFO_FAST
- TBF
- SQF
- RED

PFIFO_FAST

Questa coda è, come dice il nome, del tipo *First In-First Out*, che significa che nessun pacchetto riceve un trattamento speciale. Questa coda ha 3 così dette "bande", e all'interno di ognuna di esse viene applicata la logica FIFO.



La "colonna" zero ha la massima priorità e fino a che non si svuota le altre rimangono in attesa, allo stesso modo la colonna numero 2 non potrà svuotarsi fino a che ci sono pacchetti nella colonna numero 1.

Parametri ed uso

Per configurare questo tipo di accodamento ci serviamo di 2 parametri.

Il primo parametro è ***priomap*** che determina la priorità dei pacchetti.Il nucleo del sistema operativo legge il TOS (Type of Service) nel campo d'intestazione di un pacchetto IP, ed è in base al valore di questo campo che i pacchetti sono divisi in ranghi o bande.

```
   0     1     2     3     4     5     6     7
+-----+-----+-----+-----+-----+-----+-----+-----+
|                 |                       |     |
|    PRECEDENCE   |          TOS          | MBZ |
|                 |                       |     |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Quattro degli otto bit TOS campi sono definiti come indicato nella tabella sotto:

```
Binario Decimale Significato
----------------------------------------
1000      8     Minimizza il ritardo (md)
0100      4     Massimizza il throughput (mt)
0010      2     Massimizza affidabilità (mr)
0001      1     Minimizza il costo (mmc)
0000      0     Normale
```

Non bisogna confondere questo tipo di accodamento con uno che usa le classi, perché sebbene abbiano un comportamento simile, il pfifo_fast è senza classi e non potrete aggiungere, per esempio, un altra coda *qdisc* col comando tc.

Col comando "tcpdump -vv" verrà mostrato il valore dell'intero campo TOS di ogni pacchetto, non solo dei primi 4 bit.

```
TOS  Bits Significato             Linux Priority      Banda
------------------------------------------------------------
0x0   0   Normale                 0 Migliore sforzo   1
0x2   1   Minimizza Costo         1 Riempitore        2
0x4   2   Massimizza Affidabilità 0 Migliore sforzo   1
0x6   3   mmc+mr                  0 Migliore sforzo   1
0x8   4   Massimizza Throughput   2 Rinfusa           2
0xa   5   mmc+mt                  2 Rinfusa           2
0xc   6   mr+mt                   2 Rinfusa           2
0xe   7   mmc+mr+mt               2 Rinfusa           2
```

```
0x10  8   Minimize Delay          6 Interattivo       0
0x12  9   mmc+md                  6 Interattivo       0
0x14  10  mr+md                   6 Interattivo       0
0x16  11  mmc+mr+md               6 Interactive       0
0x18  12  mt+md                   4 Int. Rinfusa      1
0x1a  13  mmc+mt+md               4 Int. Rinfusa      1
0x1c  14  mr+mt+md                4 Int. Rinfusa      1
0x1e  15  mmc+mr+mt+md            4 Int. Rinfusa      1
```

Le prime 2 colonne contengono il possibile valore in formato sia esadecimale che decimale e dato che si tratta di un numero a 4 bit il suo valore varia da 0 a 15. Per esempio se il campo TOS contenesse il numero 15 vorrebbe dire che tutti i 4 bit sarebbero impostati ad 1 e di conseguenza il pacchetto richiederebbe la combinazione di tutte e quattro le caratteristiche descritte nella tabella precedente: Minimo Costo Monetario, Massima Affidabilità, Massimo Throughput e Minimo Ritardo.
La quarta colonna rappresenta come il kernel Linux interpreta i bits del TOS, mostrando con che priorità vengono mappati.
L'ultima colonna mostra il risultato della priomap di default. Da linea di comando una priomap apparirebbe così:

```
                              1, 2, 2, 2, 1, 2, 0, 0 , 1, 1, 1, 1, 1, 1, 1, 1
```

Questo significa che i pacchetti che il kernel interpreta come priorità 4, per esempio, verranno inseriti nella banda numero 1 del nostro accodamento FIFO_fast e priorità anche maggiori di 7 che non risultano dal mappagio TOS possono essere usate ai fini dell'accodamento.

La seguente tabella estratta dal RFC 1349 ci mostra come varie applicazioni impostano il TOS dei loro pacchetti:

```
TELNET               1000 (minimizza ritardo)
FTP
  Control            1000 (minimizza ritardo)
  Data               0100 (massimizza throughput)
TFTP                 1000 (minimizza ritardo)
SMTP
  Command phase      1000 (minimizza ritardo)
  DATA phase         0100 (massimizza throughput)
Domain Name Service
  UDP Query          1000 (minimizza ritardo)
  TCP Query          0000
  Zone Transfer      0100 (massimizza throughput)
NNTP                 0001 (minimizza costo monetario)
ICMP
  Errors             0000
  Requests           0000
  Responses          0000
```

Il secondo parametro è ***txqueuelen*** con il quale impostiamo la lunghezza della coda ed è possibile impostarlo dalla configurazione delle interfacce, tramite il comando *ifconfig* e *ip*.
Per impostare la lunghezza della coda a 10 basta eseguire il comando:

```
ifconfig eth0 txqueuelen 10
```

Non è possibile mostrare il valore di questo parametro con il comando tc.

TOKEN BUCKET FILTER

Il Token Bucket Filter (TBF) è un semplice metodo di accodamento che fa passare i pacchetti in arrivo che non superano una velocità di soglia preimpostata e con la possibilità di consentire dei piccoli eccessi rispetto a questa soglia.

Il TBF è molto preciso e non necessita di una gran capacità di calcolo, e dovrebbe essere essere preso in considerazione come prima metodo per rallentare semplicemente una interfaccia.

Questa struttura di coda è immaginabile come una secchio con un buco sul fondo per cui il flusso in uscita tende ad essere costante (a meno che il secchio non sia vuoto). Quando viene riempito troppo velocemente si riempi e l'acqua in eccesso cade ai bordi e non è più recuperabile. Allo stesso modo possiamo definire la dimensione massima della coda e il suo rate di uscita .

La sua implementazione consiste in un buffer (bucket), costantemente riempito da alcuni pezzi virtuali di informazione chiamati *tokens*, a una specifica velocità (*token rate*). Il parametro più importante del "secchio" è la sua grandezza, che è data dal numero di tokens che può immagazzinare. I tokens sono come degli slot vuoti che fluiscono all'interno del buffer e prendendosi carico dei pacchetti li fanno transitare all'interno della coda.

In questo tipo di accodamento abbiamo due tipi di flussi:

- Il flusso dei dati in arrivo
- Il flusso di token che liberi che si propongono

a seconda della velocità relativa di questi 2 flussi ci possiamo trovare in alcune situazioni:

- I dati che arrivano alla coda hanno la stessa velocità dei token. In questo caso tutti i pacchetti di dati verranno allocati nei rispettivi slot(token) e passeranno attraverso la coda (bucket) senza ritardi.
- I dati che arrivano al TBF hanno una velocità inferiore di quella con cui si liberano i token. anche in questo caso non ci saranno ritardi.
- I dati che arrivano nella coda hanno una velocità maggiore dei token. questo significa che il contenitore sarà ben presto a corto di token liberi, e ci troveremo in una situazione di "fuori limite" e i pacchetti in arrivo verranno scartati.

L'ultimo scenario è molto importante, perché consente di limitare il flusso di dati imponendo una banda massima come filtro.

La capacità del contenitore, misurata in token, non è del tutto fissa, ma consente alcune variazioni per eccesso. L'accumulazione dei token liberi potrebbe portare la capacita del contenitore un po oltre la sua soglia, e per un breve periodo di tempo si potrebbe avere una velocità in uscita un po superiore (bursts), ma in situazioni di sovraccarico i pacchetti verranno in un primo momento ritardati e poi infine scartati.

Notare che nella attuale implementazione dei token essi sono misurati in bytes non in pacchetti

Uso:

```
... tbf limit BYTES burst BYTES[/BYTES] rate KBPS [ mtu
       BYTES[/BYTES] ][ peakrate KBPS ] [ latency TIME ]
```

- burst/buffer/maxburst: indica la dimensione del secchio in bytes. Limita il numero massimo in bytes occupabile dai tokens in un determinato istante. In teoria dovrebbe essere proporzionato rispetto alla velocità che si vuole trasmettere.
- limit or latency: con la latenza indico il tempo massimo che un pacchetto può restare all'interno della coda, mentre con limit indico il numero di byte che possono restare in attesa di un token disponibile.
- rate : indica quanti tokens al secondo ho a disposizione, in pratica è la velocità massima di trasmissione.
- mpu:pacchetti al di sotto di questa dimensione non vengono accodati. La Minimum Packet Unit determina la taglia dei token per i pacchetti.
- peakrate: Se un token è disponibile e dei pacchetti arrivano, sono spediti immediatamente. Questa potrebbe non essere la tua volontà specie se hai un grosso secchio. Il peakrate indica al massimo quanto rapidamente vuoi che si svuoti il secchio.
- mtu/minburst Se avessimo un 1mbit/s di peakrate ed una velocità(rate) molto maggiore questo non sarebbe molto conveniente. Un peakrate più grande è possibile mandando più pacchetti per timertick, è questo parametro mtu/minburst ci da la grandezza di questa cache a causa delle restrizioni sul peakrate.

Esempio di TBF

Una configurazione molto utile potrebbe essere questa:

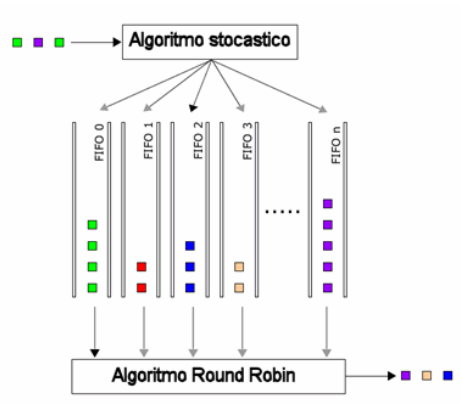
```
 # tc qdisc add dev ppp0 root tbf rate 220kbit latency 50ms burst 1540
```

perché questo tipo di accodamento potrebbe farci comodo? Se avessimo un dispositivo che ci connette ad internet con una "coda" molto capiente, quale potrebbe essere un router con un modem DSL, e vogliamo comunicare con Skype o qualche altro sistema di comunicazione vocale, il nostro upload distruggerebbe l'interattività. Questo perché la coda capiente del router verrebbe riempita dai dati in upload e i dati della comunicazione vocale verrebbero accodati e probabilmente spediti in ritardo, ma questo in una comunicazione realtime è inaccettabile.
Quindi quello che faccio io è di limitare i dati in uscita dal mio pc così da non dover usare l'accodamento nel router. Nell'esempio è stata impostata una velocità di upload di 220kbit, nel caso vogliate usare questo comando aggiornate il numero a valore di upload della vostra connessione ad internet meno una certa percentuale. Se avete un modem molto veloce potete incrementare un po il valore di 'burst'.

Stochastic Fairness Queueing

Stochastic Fairness Queueing (SFQ) , tradotto sarebbe metodo di accodamento nel giusto ordine, è una semplice applicazione di metodi per la gestione del traffico della famiglia dei metodi di parità del controllo.
È meno accurato rispetto agli altri, ma è molto più veloce (meno calcoli), con la caratteristica dello stesso trattamento per tutti i pacchetti.
La principale caratteristica del metodo SFQ è che tratta allo stesso modo indistintamente i dati con protocollo TCP e UDP della sessione corso.
Individua quindi ogni sessione TCP o UDP e divide stocasticamente il traffico in flussi, e ad ogni flusso viene assegnata una qdisc FIFO (coda) alla quale a turno viene data la possibilità di trasmettere.
L'algoritmo Round Robin, si occupa di inviate a turno da ogni coda creata un certa quantità di dati (quantum), e questo si traduce in una parità di condotta, ed esclude la possibilità che un flusso sia preferito ad un altro.
Questo comportamento "giusto" fa in modo che nessuna singola "conversazione" prevalga sulle altre. L'SFQ è chiamato "Stocastico" perché non crea realmente una coda per ogni sessione, ma ha un algoritmo che smista il traffico su un numero limitato di code usando un algoritmo di rimescolamento casuale (hash).

A causa di quest'ultimo, sessioni multiple potrebbero finire nella stessa coda, per cui queste sessioni avrebbero la metà delle possibilità di trasmettere, e quindi metà della velocità rispetto alle altre, che hanno una coda per conto loro.
Per prevenire questo l'SFQ cambia l'algoritmo di hash abbastanza spesso così che due sessioni verrebbero dirette nella stessa coda (collisione) solo per un po di tempo.



Bisogna notare che l'SFQ è utile solo se l'interfaccia d'uscita a cui viene applicato è veramente satura, altrimenti non ci sarebbe alcun vantaggio dato che non si noterebbero gli effetti. Quindi il consiglio è quello di utilizzarlo insieme ad altri metodi di accodamento, in modo da sfruttare i vantaggi di entrambi i metodi.

Parametri ed uso

Il metodo SFQ è facilmente configurabile:

Uso: ... sfq [perturb SECS] [quantum BYTES] [limit]

perturb: parametro per l'algoritmo di confusione (tempo di miscelazione). Stabilisce dopo quanti secondi vine modificato l'algoritmo. Se il parametro non è impostato l'algoritmo non verrà mai cambiato e questo è sconsigliato. Un valore accettabile è 10 secondi.

quantum: numero di bytes trasmittibili prima di cedere la mano. Di default è il valore del' MTU (la grandezza massima per un pacchetto), non bisogna mai impostare questo valore in maniera tale che sia inferiore all'MTU.

limit : numero totale di pacchetti che possono essere accodati dall' SFQ (dopo il quale verranno scartati).

Esempio di configurazione

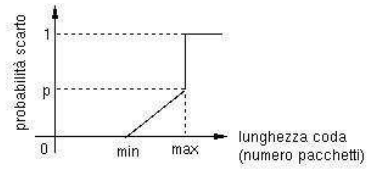
Per impostare l'SFQ sulla interfaccia del nostro pc collegata al modem o al router lanciamo questi comandi:

```
# tc qdisc add dev ppp0 root sfq perturb 10
# tc -s -d qdisc ls
qdisc sfq 800c: dev ppp0 quantum 1514b limit 128p flows 128/1024 perturb 10sec
Sent 4812 bytes 62 pkts (dropped 0, overlimits 0)
```

Il secondo comando mostra i parametri del metoto che abbiamo impostato col primo comando.
Il numero *800c:* è l'identificativo automaticamente assegnato alla qdisc, il parametro *limit* è impostato a 128 che è la grandezza della coda (totale). Ci sono poi 1024 *hashbuckets* (code FIFO) disponibili, di cui se ne possono usare 128 alla volta, dato che questa è la dimensione massima della coda. Ogni 10 secondi l'algoritmo viene riconfigurato.

RED

Una coda Red (random Early Detection) può essere vista come una variante di quella FIFO in quanto in uscita i pacchetti sono spediti allo stesso modo mentre ciò che varia è il meccanismo di accodamento. Innanzi tutto bisogna definire due soglie una minima e una massima. Per capire come lavorano aiutiamoci con un grafico:



p: probabilità di scarto massima
min: soglia minima
max: soglia massima

Quando la coda ha una lunghezza maggiore o uguale della soglia massima qualunque pacchetto che arriva viene scartato, mentre quando ci si trova tra le due soglie inizia uno scarto casuale dei pacchetti in arrivo con una probabilità crescente tanto più la coda si riempie.

Uso:

```
... red limit BYTES min BYTES max BYTES avpkt BYTES burst PACKETS probability PROBABILITY bandwidth KBPS [ ecn ]
```

In pratica questo tipo di algoritmo si usa solitamente per grossi volumi di traffico,ad esempio collegamenti a 100Mbps, con i quali bisogna usare delle code molto capienti, questo perché il traffico è tanto e dobbiamo garantire che le trasmissioni vadano a buon fine, quindi diminuire le ritrasmissioni. A questo scopo è stato introdotto quelsto algoritmo che evita la congestione del traffico, controllando lo stato della coda utilizzando le due soglie. In questo modo le connessioni che occupano più banda hano più probabilità di essere "tagliate" evitando così la congestione globale.
Nella realtà questo algoritmo viene associato all'ECN (Explicit Congestion Notification) per notificare agli host situazioni di congestione della rete marcando i pacchetti in modo opportuno. Gli host che supportano l'ECN saranno saranno in grado di auto limitarsi riducendo al congestioen della rete (RFC 3168).

Per approfondimenti leggere questo articolo.

Le qdisc Classful sono molto utili se si dispone di diversi tipi di traffico, i quali dovrebbero essere trattati in maniera diversa.

Uno dei metodi classful è il così detto "*cBq*", "class base queue", che risulta essere il più famoso, ma solo percheè risulta essere il più antico e complesso.

IL FLUSSO DI DATI NEI METODI CLASSFUL

Quando entra del traffico dati entra in un accodamento classful, deve essere selezionato ed inviato in qualche classe, il dato ha quindi bisogno di essere "*classificato*". Per sapere che cosa a che fare con un pacchetto, si ricorre ai "*filtri*". E' importante sapere che i filtri sono usati da una qdisc, e non il contrario!

I filtri connessi a tale qdisc ci forniscono una decisione e che la qdisc utilizza per accodarli in una classe. Ogni sottoclasse può a sua volta ricorrere ad altri filtri per ulteriori selezioni e classificazioni. In caso contrario, la classe accoda il pacchetto alla qdisc alla quale è connessa.

Queste qdisc classful inoltre possono eseguire uno shaping dei dati, che dopo essere stati classificati possono essere ritardati e inviati ad una velocità impostata. Come nei casi classless, avremo bisogno di questa soluzione, nei casi in cui si dispone di una interfaccia ad alta velocità (ad esempio, Ethernet) connessa un dispositivo più lento (un modem via cavo).

LA GERARCHIA NELLE QDISC: ROOTS, HANDLES, SIBLINGS E PARENTS

Ogni interfaccia ha in uscita una "*root* qdisc" (root = radice). Di default è un tipo di accodamento *pfifo_fast*, che non una le classi. Per ogni qdisc o classe è assegnato un *handle* (identificativo), che può essere usato per riferirsi a quella particolare coda o classe nel resto della configurazione. Oltre ad una qdisc in uscita una interfaccia può averne una in ingresso, che ne regola il traffico (policing).

Gli *handles* di queste qdiscs sono composti da due parti separate dal simbolo ":" e queste sono: :.
Solitamente la root qdisc è '1:', che vuol dire '1:0'. Il numenro *minore* di una qdisc è sempre 0.

Le classi necessitano d'avere lo stesso numero *maggiore* come loro *genitore*. Questo numero deve essere unico per una configurazione d'ingresso o una d' uscita. Il numero *minore* deve essere unico per una qdisc e le sue classi.

Vediamo di ricapitolare la situazione forse un po confusa facendo un esempio:

```
                1:   root qdisc
                 |
                1:1    child class
              /  |  \
             /   |   \
            /    |    \
           /     |     \
```

```
             1:10  1:11  1:12   child classes
              |      |     |
              |     11:    |    leaf class
              |            |
              10:         12:   qdisc
             /   \       /   \
          10:1  10:2   12:1  12:2   leaf classes
```

Come detto prima i nodi di questo albero che non hanno il numero minore (è sottinteso ":0") sono delle qdisc, mentre gli altri sono classi.
Tutte le classi terminano con una qdisc, dove non appare esplicitamente questo è sottinteso e la qdisc sarà una semplice fifo. Il primo numero ("maggiore") di un handle di classe rappresenta "*a cosa*" la classe è collegata, questo significa che una classe *10:2* sarà collegata alla coda *10:* o *10:0*.

Ad ogni nodo è collegato un filtro attraverso il quale si seleziona e si smista il traffico negli altri nodi.

Un pacchetto ad esempio può, attraverso una serie di filtri in ogni nodo, fluire nella catena:

1: -> 1:1 -> 1:12 -> 12: -> 12:2

ed andare a finire nella qdisc collegata alla classe 12:2.

Comunque è anche possibile che un filtro applicato alla root mandi il pacchetto direttamente nella classe desiderata *12:2*:

1: -> 12:2

Come i pacchetti vengono fatti uscire dall'accodamento (dequeued).

Quando il kernel ha bisogno di inviare i pacchetti accodati all'interfaccia di destinazione si rivolge alla *root qdisc* "1:" facendo una richiesta di "*dequeue*", questa è passata alla 1:1, che a sua volta è passata alla 10:, 11: e 12:, ognuna fa la richiesta alle suoi discendenti. In questo caso il kernel ha bisogno di ripercorrere tutto l' "albero" perché solo il nodo 12:2 contine il pacchetto.

In poche parole, le classi "figlie" possono parlare solo con i loro genitori e mai con una interfaccia, e quindi solo la qdisc radice può ricevere ordini dal kernel e disaccodare un pacchetto.

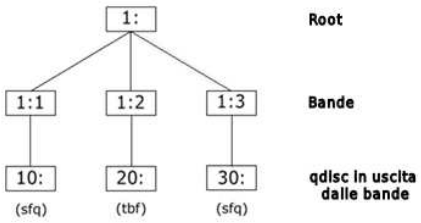
L'esito di questo sistema è che le classi figlie non possono mai disaccodare i pacchetti più velocemente delle classi da cui discendono. Questo in effetti è quello che si vuole ottenere, infatti , se avessimo una SFQ come qdisc genitore, la quale non fa shaping, ma solo scheduling, e una TBF, in grado di ritardare i pacchetti come figlia, allora ci troveremo in una situazione ottimale, dato che la TBF discaccoderebbe i pacchetti più lentamente, ad una velocità impostata da noi e la SFQ penserebbe a miscelarli con l'altro traffico restante. Se ci trovassimo nella situazione contraria tutto il lavoro di scheduling della SFQ sarebbe in parte vanificato dal rallentamento a monte nella TBF, che potrebbe non essere quello che vogliamo.

La qdisc PRIO

La qdisc PRIO non fa shaping (ritardo pacchetti), ma suddivide il traffico, tramite dei filtri da configurare, in classi interne. E' possibile considerare la PRIO qdisc una specie di pfifo_fast più evoluta, dove le "bande" della fifo questa volta sono delle vere e proprie classi.

Graffico PRIO con le tre bande di default

Quando un pacchetto è accodato nella PRIO, viene classificato tramite i filtri che sono stati impostati in fase di configurazione. Di default sono create solo 3 classi, le quali contengono delle semplici code FIFO senza altre strutture, la differenza tra la questa e la pfifo_fast, senza classi, e che questa configurazione di base può essere modificata a nostro piacimento, cosa che nel metodo classless non potevamo fare.



Schema PRIO con le qdisc in uscita personalizzate

Per disaccodare i pacchetti viene richiesto inizialmente alla classe *:1* poi a tutte le altre. In pratica vengono svuotate prima le classi basse poi quelle con numerazione più alta.

Questo tipo di comportamento serve a dare priorità ad un certo tipo di traffico usando non soltanto il TOS flag ma anche tutti i tipi di filtri che si possono creare al comando tc, diversamente da quanto accadeva col metodo classless.

Non avviene alcuno shaping in uscita, come per la qdisc SFQ, e quindi ha senso usarlo solo se il traffico in gioco è maggiore di quello che può essere inviato, altrimenti non si vede nessun vantaggio a dare le priorità ai pacchetti.

Formalmente la qdisc PRIO è un metodo Work-Conserving scheduler.

Uso e parametri PRIO

Uso:

```
... prio bands NUMBER priomap P1 P2..
```

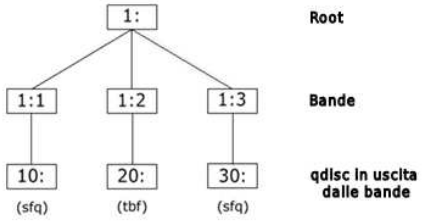
I parametri disponibili con il metodo priorità sono 2:

- *bands* – vale a dire, il numero di classi che vengono create
- *priomap* – se non sono stati forniti dei filtri per la classificazione del traffico allora il metodo PRIO usa la priorita TC_PRIO per selezionare in quale banda allocare i pacchetti.
  Questo parametro funziona come per il metodo *pfifo_fast* .

Le "bande" sono classi, e sono chiamate *:1*, *:2* e *:3* di default, così se la qdisc PRIO è chiamata *12* allora queste avranno come identificativo *12:1*, *12:2* e *12:3*. Quella chiamata *12:1* avrà una priorità maggiore.

Esempio di configurazione

Riprendendo lo schema della figura qua sotto



Vorrei creare una qdisc che mandi sulla qdisc *30:* tutto il traffico non classificato (Bulk traffic), mentre mandi quello interattivo sulla *20:* o sulla *10:*.

I comandi sono:

```
# tc qdisc add dev eth0 root handle 1: prio
## Questo crea istantaneamente le classi 1:1, 1:2, 1:3

# tc qdisc add dev eth0 parent 1:1 handle 10: sfq
# tc qdisc add dev eth0 parent 1:2 handle 20: tbf rate 20kbit buffer 1600 limit 3000
# tc qdisc add dev eth0 parent 1:3 handle 30: sfq
```

Adesso vediamo cosa abbiamo appena creato:

```
# tc -s qdisc ls dev eth0

qdisc sfq 30: quantum 1514b
Sent 0 bytes 0 pkts (dropped 0, overlimits 0)

qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
Sent 0 bytes 0 pkts (dropped 0, overlimits 0)

qdisc sfq 10: quantum 1514b
Sent 132 bytes 2 pkts (dropped 0, overlimits 0)

qdisc prio 1: bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
Sent 174 bytes 3 pkts (dropped 0, overlimits 0)
```

Per ogni nodo appare la sua attuale configurazione e i pacchetti che vi transitano.

CBQ (class-based queuing)

E' il tipo di accodamento più completo a disposizione. Permette di sagomare perfettamente il tipo di traffico che dovrà trattare . Questa qualità si paga con una complessità che è di gran lunga maggiore delle altre qdisc.

Il metodo CBQ è anche lui uno shaper del traffico è dovrebbe lavorare così: se abbiamo una banda disponibile d'uscita di 10mbit/s che vogliamo limitare a 1 mbit/s allora lo shaper CBQ lavora in modo tale da lasciare all'interfacia d'uscita libera per il 90% del tempo.

Quindi CBQ lavora in modo tale da essere sicura che il collegamento risulti libero per un tempo opportuno tanto da abbattere il rate trasmissivo fino alla limitazione configurata. Per fare ciò essa calcola il tempo medio ipotetico, *idle*, che dovrebbe passare tra i pacchetti. Durante le operazioni , invece, viene misurato realmente questo tempo (*idletime*), usando una media mobile esponenziale (EWMA) che considera più importante l'informazione relativa ai pacchetti più recenti. La differenza tra il tempo ideale e quello effettivo è chiamato *avgidle.* Se il tempo effettivo dei dati in arrivo è maggiore di quanto calcolato (avgidle > 0) il collegamento accumula crediti trasmissivi fino ad un tetto massimo (maxidle), dato che se non vi fosse questo limite dopo ore di idle il link dovrebbe permettere una banda infinita.
Mentre se questo tempo effettivo è minore di quello previsto, la CBQ chiude i battenti per un po\' essendo in sovraccarico (avgidle < 0). In questa situazione la CBQ dovrebbe stozzare il canale per un tempo pari al *avgidle* calcolato quindi fare passare un pacchetto e richiudelo nuovamente.
In realtà questi comportamenti critici sono gestiti tramite i parametri maxburst e minburst.

Oltre al comportamento di shaper descritto prima il metodo CBQ funziona anche come la PRIO, riuscendo a differenziare il traffico usando delle priorità.
Infatti ogni volta che c'è la richiesta da parte del layer hardware di spedire un pacchetto, un algoritmo di round robin "pesato" (weighted round robin = WRR) inizia a mandare via i pacchetti iniziando da quelli identificati con una classe di priorità più bassa.
Dopo che una classe ha mandato un certo numero di bytes, viene permesso alla successiva di inviare i propri.

Parametri ed uso della qdisc CBQ

Uso:

```
... cbq bandwidth BPS rate BPS maxburst PKTS [ avpkt BYTES ]
    [ minburst PKTS ] [ bounded ] [ isolated ]
    [ allot BYTES ] [ mpu BYTES ] [ weight RATE ]
    [ prio NUMBER ] [ cell BYTES ] [ ewma LOG ]
```

```
    [ estimator INTERVAL TIME_CONSTANT ]
    [ split CLASSID ] [ defmap MASK/CHANGE ]
```

- avpkt : dimensione media dei pacchetti espressa in bytes. Ne abbiamo bisogno per calcolare il maxidle, usando anche il maxburst.
- bandwidth : la banda fisica della interfaccia, serve per il calcolo dei tempi di idle.
- Cell : Determina la granularità dei pacchetti nei calcoli delle tempi di trasmissione. Il tempo che un pacchetto trasmesso impiega per uscire dall'interfaccia potrebbe crescere in maniera discontinua. Solitamente questo parametro viene impostato a "8" (deve essere sempre una potenza di 2), questo implica che ai fini del calcolo della velocità di trasmissione un pacchetto di 800 bytes e uno di 806 impiegano lo stesso tempo.
- maxburst : questo numero di pacchetti è usato per calcolare il massimo tempo prima che avgidle crolli a zero (*maxidle*). Più e alto più CBQ risulta tollerante ai burst di pacchetti. Notare che non è possibile impostare direttamente il maxidle ma solo il maxburst.
- minburst: CBQ necessita di strozzare il canale trasmissivo in caso di overlimit. Quando si arriva al *avgidle* il canale viene riaperto e passa un pacchetto o più a seconda di quanto qui specificato. Più pacchetti passano, più tempo dovrà passare per una riapertura del canale (offtime).
- minidle: se avgidle è un numero negativo siamo in overlimits e bisogna aspettare fino a che non cresce fino a rendere possibile la spedizione di un pacchetto. Qui si definisce il valore limite , in basso, che può asssumere avgidle. Comunque è un numero negativo quindi 10microsecondi significa -10microsecondi.
- mpu: dimensione minima dei pacchetti.
- rate: rate trasmissivo desiderato.
- isolated: si specifica che una classe non può ne prestare banda alle altre classi. Il contrario di questa situazione si ha impostando lo *sharing* tra le classi.
- bounded: si specifica che una classe non può chiedere in prestito banda. Il contrario è lo stato di *borrow* che consente ad una classe di chiedere banda in prestito da chi può fornirla.
- allot: quando la CBQ in uscita deve far uscire i pacchetti proverà a svuotare tutte le qdisc nelle classi in ordine di priorità. A turno tutte spediranno i loro dati, ma solo una quantità limitata, definita dal parametro **allot** appunto.
- prio: in riferimento a quanto detto per il parametro allot, la CBQ può agire come una qdisc prio, cioè le qdisc delle classi interrogate a turno in ordine di priorità svuotano tutto il loro contenuto prima di cedere la mano alla successiva, al contrario di quanto succedeva col parametro allot.
- weight: questo parametro aiuta il funzionamento dell'algoritmo Weighted Round Robin. Ogni classe ha una certa probabilità di inviare a turno i prorpi dati. Se avessimo classi con più banda di altre, avrebbe senso permettergli di inviare più dati delle altre ad ogni turno. Una CBQ aggiunge un peso per ogni classe, in modo da normalizzarne il valore totale e in modo che l'utente possa impostare dei valori arbitrari, questo significa che solo il rapporto tra questi valori è importante, non i valori in se.
  I valori dei pesi così normalizzati sono moltiplicati per il parametro *allot* per determinare quanti dati possono essere inviati ad ogni turno da quella particolare classe di traffico.

Esempio di uso di una CBQ

Prendiamo questo esempio di configurazione:

```
              1:          root qdisc
              |
             1:1          classe figlia
            /   \
           /     \
         1:3     1:4       classi foglie
          |       |
         30:     40:       qdiscs
        (sfq)   (sfq)
```

Il problema da risolvere è questo: il traffico di un webserver deve essere limitato a 5mbit e il traffico SMTP a 3mbit. Insieme però loronon possono superare i 6mbit. Abbiamo a disposizione una scheda di rete a 100mbit e le classi possono prestarsi banda tra loro.

```
# tc qdisc add dev eth0 root handle 1:0 cbq bandwidth 100Mbit         \
  avpkt 1000 cell 8
# tc class add dev eth0 parent 1:0 classid 1:1 cbq bandwidth 100Mbit  \
  rate 6Mbit weight 0.6Mbit prio 8 allot 1514 cell 8 maxburst 20      \
  avpkt 1000 bounded
```

Questa serie di comandi installa la *root* e la classe *1:1*. Quest'ultima è stata impostata come *bounded*, così la larghezza di banda totale non potra superare i 6mbit.

```
# tc class add dev eth0 parent 1:1 classid 1:3 cbq bandwidth 100Mbit  \
  rate 5Mbit weight 0.5Mbit prio 5 allot 1514 cell 8 maxburst 20      \
  avpkt 1000
# tc class add dev eth0 parent 1:1 classid 1:4 cbq bandwidth 100Mbit  \
  rate 3Mbit weight 0.3Mbit prio 5 allot 1514 cell 8 maxburst 20      \
  avpkt 1000
```

Con questi comandi si definiscono le 2 classi foglie. Esse hanno due pesi differenti nella configurazione della velocità. Sono entrambe non *bounded* e connesse alla stessa classe 1:1, che invece è *bounded*, in questo modo la somma delle bande delle 2 classi foglia non supererà mai i 6mbit, come volevano le specifiche. I loro classid inoltre hanno entrambe lo stesso genitore come numero .

```
# tc qdisc add dev eth0 parent 1:3 handle 30: sfq
# tc qdisc add dev eth0 parent 1:4 handle 40: sfq
```

Entrambe le classi sono collegate a delle qdisc FIFO di default, in questo caso invece queste sono rimpiazzate da una coda SFQ, così ogni flusso è trattato

equamente.

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
  sport 80 0xffff flowid 1:3
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
  sport 25 0xffff flowid 1:4
```

Con questi comandi, attaccheremo dei filtri direttamentealla root, in modo che il traffico venga instradato nella giusto nodo.

Notare che noi abbiamo usato il comando "tc class add" per aggiungere e creare delle classi dentro una qdisc, e il comando "tc qdisc add" per aggiungere una qdiscs alle classi.

Tutto il traffico che non rientra dentro la classificazione dei filtri sarà inviato dentro la 1:0 (root) e sarà illimitato.
Se il traffico SMTP+web insieme cercherà di superare la banda limite di 6Mbit/s, la laghezza di banda sarà divisa tra i due nel rapporto 5/8 per il webserver e 3/8 per il mail server, come previsto dal parametro weight.
Con questa configurazione possiamo anche dire che sempre un minimo di 5/8 * 6 mbit = *3.75 mbit*.

Altri parametri per la CBQ : split & defmap

La qdisc CBQ oltre a dirigere il traffico sulle vaie classi offre dei meccanismi alternativi, quali *defmap* e *split*.
Questi agiscono sul Type of Service (TOS) per selezionare i pacchetti e instradarli nella classe impostata. Quando la CBQ ha bisogno di dedurre dove deve accodato un pacchetto, controlla se questo nodo è un "split node". Se è così, una delli suoi sotto-accodamenti sarà indicato come destinazione per i tutti i pacchetti con una certa configurazione di priorità (TOS).
Questo è un metodo molto veloce per classificare i pacchetti.

Una defmap si *ff* (hex) corrisponde a qualsiasi bit, una map di *0* non ha nessun mach. Per esempio:

```
# tc qdisc add dev eth1 root handle 1: cbq bandwidth 10Mbit allot 1514 \
  cell 8 avpkt 1000 mpu 64

# tc class add dev eth1 parent 1:0 classid 1:1 cbq bandwidth 10Mbit    \
  rate 10Mbit allot 1514 cell 8 weight 1Mbit prio 8 maxburst 20        \
  avpkt 1000
```

Questo è il preambolo standard per la CBQ preamble.

La *Defmap* si riferisce ai bit TC_PRIO, definiti in questo modo:

```
TC_PRIO..          Num  Corresponds to TOS
------------------------------------------------
BESTEFFORT         0    Maximize Reliablity
FILLER             1    Minimize Cost
BULK               2    Maximize Throughput (0x8)
INTERACTIVE_BULK   4
INTERACTIVE        6    Minimize Delay (0x10)
CONTROL            7
```

La numerazione TC_PRIO.. corrisponde ai bits del TOS contati da destra. Per maggiori dettagli su come questi bit sono convertiti in priorità vedere il paragrafo relativo alla *pfifo_fast*.

Le classi per il traffico di tipo interattivo e bulk:

```
# tc class add dev eth1 parent 1:1 classid 1:2 cbq bandwidth 10Mbit     \
  rate 1Mbit allot 1514 cell 8 weight 100Kbit prio 3 maxburst 20        \
  avpkt 1000 split 1:0 defmap c0

# tc class add dev eth1 parent 1:1 classid 1:3 cbq bandwidth 10Mbit     \
  rate 8Mbit allot 1514 cell 8 weight 800Kbit prio 7 maxburst 20        \
  avpkt 1000 split 1:0 defmap 3f
```

La "*split qdisc*" è *1:0*, che dove viene fatta la selezione del traffico.
La classe foglia *1:2* ha come defmap il valore *C0* che in binario sarebbe *11000000*, la classe *1:3* invece ha come defmap il valore *3F* cioè *00111111*,
L'insieme delle due defmap compre tutti i possibili casi del valori di TOS. Il valore di TOS è un numero a 8 bit, la defmap è una masckera per questo valore.
Nel primo esempio la classe riconosce come propri tutti i pacchetti che hanno il 6°e 7°bit del loro TOS ad 1 , il quale corrisponde al traffico "interattivo" e "controllo". Il Secondo caso corrisponde a tutti i casi restanti.

Il nodo *1:0* ha quindi una tabella di instradamento come questa:

```
priorità    manda a
     0        1:3
     1        1:3
     2        1:3
     3        1:3
     4        1:3
     5        1:3
     6        1:2
     7        1:2
```

Potremo anche modificare direttamente questa tabella con "*change mask*", che indica esattamente quali priorità modificare. Per esempio, per aggiungere il tipo di traffico "best effort" alla classe *1:2* :

```
# tc class change dev eth1 classid 1:2 cbq defmap 01/01
```

La mappa delle priorità in 1:0 adesso sarà così:

```
priorità    send to
    0        1:2
    1        1:3
    2        1:3
    3        1:3
    4        1:3
    5        1:3
    6        1:2
    7        1:2
```

HTB (HIERARCHICAL TOKEN BUCKET)

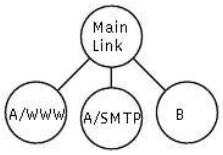
L'HTB è più comprensibile, intuitivo e veloce del metodo di accodamento CBQ. Entrambi i metodi CBQ e HTB aiutano nel controllo della banda in uscita su di un link. Entrambi consentono l'uso di un unico link fisico simulando dei link virtuali più lenti in cui inviare il traffico dati classificato. In entrambi i casi bisogna specificare come fare questa suddivisione e quali pacchetti metterci dentro. Solo che la HTB lo fa in maniera più intuitiva.

Nota : *kbps* significa kilobytes per secondo e *kbit* significa kilobits

CONDIVISIONE DELLA CONNESSIONE

Problema: Abbiamo 2 clineti, A e B, entrambi connessi ad internet via eth0. Noi vogliamo allocare 60 kbps a B e 40 kbps ad A. Inoltre vogliamo suddividere la larghezza di banda di A in modo che 30kbps siano per il traffico WWW e 10kbps per tutto il resto. La banda inutilizzata da una classe può essere usata dalle altre che ne hanno bisogno (in proporzione alle loro bande impostate in fase di configurazione).

L'HTB assicura che il servizio riservato per ogni classe di traffico sia garantito per un valore minimo. Se il traffico fosse non impegnasse tutta la banda assegnatagli, la banda rimansta (in eccesso) è distribuita alle altre classi che en fanno richiesta.



I differenti tipi di traffico nella HTB sono rappresentati da classi, vedi il graficoo di sopra.
Vediamo che comandi usare:

```
tc qdisc add dev eth0 root handle 1: htb default 12
```

Questo comando attacca all'interfacia eth0 un metodo di accodamento HTB assegnandogli come identificativo *1:* . Questo è solo un nome con il quale ci riferiremo più tardi a questo nodo. Il "*default 12*" significa che qualsiasi traffico non classificato verrà assegnato alla classe *1:12*.

Nota: In generale (non solo per l' HTB ma anche per tutte le qdisc e le classi nel controllo del traffico), gli handles o identificativi sono scritti nella forma x:y dove x è un intero che identifica la qdisc e y è un intero che identifica una classe discendente dalla qdisc. L'identificativo per una qdisc deve avere zero come valore in y, mentre per una classe deve essere diverso da zero. Il valore "1:" scritto sopra è come se fosse "1:0" dove però lo zero è sottinteso.

```
tc class add dev eth0 parent 1: classid 1:1 htb rate 100kbps ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:10 htb rate 30kbps ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:11 htb rate 10kbps ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:12 htb rate 60kbps ceil 100kbps
```

La prima linea crea la classe "root", *1:1* discendente dalla qdisc "root" *1:* .
Una classe è di tipo "root" se è discendente dalla qdisc htb. Una classe root, come altre classi sotto una HTB, consente ai propri "figli" di prendere in prestito banda tra loro, ma una classe root non può scambiare banda con un altra classe root.
Ad esempio, noi potremo avere anche altre classi collegate alla qdisc root, ma leccesso di banda di una non potrebbe essere sfruttata dalle altre. Nel caso noi volessimo che tra le classi ci fosse uno scambio di banda, allora dovremo creare un'altra classe superiore, che comprenda quelle tre, la quale diventerebbe la classe di root, mentre le altre , diventate classi figlie potrebbero prestarsi banda.
Questo è quello che abbiamo definito con gli ultimi tre comandi, cioè una classe root e tre classi figlie(*1:10*,*1:20* e *1:30*) ad essa collegate.
Il parametro *ceil* è spiegato dopo.

Nota: Qualcuno potrebbe chiedersi come mai in tutti i comandi viene ripetuta la parte "*dev eth0*" dato che è già stato fornito l'identificativo del nodo genitore?
La ragione è semplice, gli identificativi sono unici all'interno di una interfaccia, per esempio eth0 ed eth1 potrebbero avere entrambe al loro interno una classe chiamata *1:1* e se non specifichiamo a quale interfaccia ci stiamo riferendo la corrispondenza tra identificativo e nodo non sarebbe biunivoca.

Se all'utente A corrispondesse l'IP 1.2.3.4 e volessimo stabilire, secondo le specifiche, quali pacchetti inviare nelle classi, allora useremo i seguenti comandi:

```
tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 \
   match ip src 1.2.3.4 match ip dport 80 0xffff flowid 1:10
tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 \
   match ip src 1.2.3.4 flowid 1:11
```

Questo non è specifico per questo tipo di accodamento, ma vale per la classificazione del traffico in generale, quindi per approfondimenti rimandiamo alla sezione relativa ai filtri.
In pratica abbiamo collegato alla qdisc root due filtri, i quali cercando una corrispondenza tra i pacchetti e le regole definite al loro interno, instradano il traffico nelle varie classi.

In teoria ce ne sarebbe un terzo, ma è il famoso comportamento di default che abbiamo definito alla creazione della qdisc, infatti, tutti i pacchetti che non corrispondono a nessun filtro (qualsiasi pacchetto che non ha come sorgente 1.2.3.4) vengono di default inviati alla classe figlia *1:12*.

Eventualmente possiamo anche attaccare alle classi definite in precedenza delle qdisc diverse da quella di default (pfifo):

```
tc qdisc add dev eth0 parent 1:10 handle 20: pfifo limit 5
```

```
tc qdisc add dev eth0 parent 1:11 handle 30: pfifo limit 5
tc qdisc add dev eth0 parent 1:12 handle 40: sfq perturb 10
```

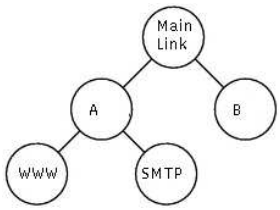
questo è tutto ciò di cui abbiamo bisogno per usare una HTB.
Vediamo cosa accade quando cerchiamo di inviare pacchetti in ogni classe ad una velocià di 90kbps per ognuna e dopo interrompere questo flusso una classe per volta:

- Quando su tutte viene riversato un flusso di 90kbps, molto maggiore del loro rate, ognuna regolarizza il flusso a seconda della velocità che gli era stata imposta: la classe: *1:10* a 30kbps, la *1:11* a 10kbps e la *1:12* a 60kbps.
- Quando la classe A *1:10* smette di riceve traffico WWW, si libera la sua banda di 30kbps, la quale viene ridistribuita con un rapporto 1/6 (10kbps/60kbps) tra le altre 2, quindi la B *1:12* avrà ora una velocità di 30*5/6+60=85kbps e l'altra 30/6+10=15kbps.
- Se ora fermassimo questi 2 flussi (*1:11* e *1:12*) e facessimo riprendere il traffico WWW della classe A, allora la banda disponibile per quest'ultima sarebbe la sua velocità base 30kbps più le altre cioè 10kbps e 60kbps, cioè avrebbe un rate di 100kbps, che non viene raggiunto in uscita perché il traffico in ingresso e di solo 90kbps.

Vediamo ora di parlare del concetto di **quantums**. Quando più classi vogliono prestarsi banda ad ognuna viene dato solo un certo numero di bytes prima che venga servita un'altra classe in competizione. Questo numero è chiamato quantum. Quindi quando alcune classi entrano in competizione per avere la banda di un altra, questa gli viene assegnata in proporzione al loro quantum. E' importante capire che per operazioni "precise" è necessario che questo quantum sia il più piccolo possibile e allo stesso tempo maggiore del MTU.
Normalmente non si ha necessità di impostare manualmente questo valore dato che la HTB assegna ad ogni classe dei valori precalcolati. Infatti essa calcola che il quantum di una classe (quando viene creata o modificata) in relazione alla sua velocità (rate) e ad altri parametri.

CONDIVISIONE GERARCHICA

Ritornando all'esempio di prima A e B sono due clienti separati, il primo che paga per 40k e il secondo per 60k. Quando la banda di A sul suo traffico WWW si libera, probabilmente A vorrebbe che fosse dedicata al suo traffico non classificato, piuttosto che andare a B.



Il problema è risolto con una gerarchia di classi come mostrato nella figura qui sopra. Il cliente A è adesso espicitamente rappresentato da una sua propria classe, e quindi tramite questa potrà richiedere tutta la sua banda (40kbps) e ridistribuirla come meglio crede tra le sue sottoclassi. Al primo livello avremo la classe root, poi le clssi che chiameremo "interne" e infine le classi foglia.
Assegneremo 40kbps al cliente A, che a sua volta avrà 2 sotto classi una per il traffico WWW da 30kbps ed un'altra da 10kbps per il restante traffico:

```
tc class add dev eth0 parent 1: classid 1:1 htb rate 100kbps ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:2 htb rate 40kbps ceil 100kbps
tc class add dev eth0 parent 1:2 classid 1:10 htb rate 30kbps ceil 100kbps
tc class add dev eth0 parent 1:2 classid 1:11 htb rate 10kbps ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:12 htb rate 60kbps ceil 100kbps
```

Adesso quando il traffico WWW del cliente A si ferma, la sua banda è assegnata al restante traffico di A che passerà da 10kbps a 40kbps.
Se A avesse comunque un traffico inferiore a 40kbps la sua banda in eccesso verrebbe passata a B.

RATE CEILING (LIMITE DI VELOCITÀ)

Il parametro **ceil** rappresenta la massima banda che una classe può usare. Questo limite, come la larghezza di banda riservata alla classe, può essere preso in prestito. Il tetto massimo di default è la stessa larghezza di banda, infatti, questo è il motivo per cui bisogna specificarlo (nell'esempio precedente altrimenti non avremo potuto effettuare scambi di banda).

Rate = Banda garantita

Ceil = Limite massimo di banda

Se nella configurazione di prima cambiassimo per la classe 1:2 (A - WWW) e la 1:11 (A - altro) il parametro *ceil* da 100kbps ai valori ripettivamente di 60kbps e 20kbps, avremo alcune differenze di comportamento:

- Al fermarsi del traffico WWW su A il restante traffico (*1:11*) non guadagnerebbe la banda liberata dalla *1:2* perché la *1:11* è limitata dal suo parametro di ceil a 20kbps. La sua banda invece verrebbe prestata a B.
- Nel caso B non avesse più traffico, tutta la sua banda dovrebbe essere trasferita ad A, ma dato che ora per A ci sono dei limiti di banda, può usarne al massimo solo 60kbps, il resto rimarrebbe inutilizzato.
  Questa caratteristica potrebbe essere utile ad un service provider, dato che potrebbero voler limitare la connessione per un servizio ad un loro utente indipendentemente dal fatto che gli altri utenti utilizzino o meno lo stesso servizio.

Notare che le classi root non possono prendere in prestito della banda, quindi non si può specificare il parametro ceil per loro, dato che questo coincide sempre con la banda assegnatagli.

Il parametro ceil deve essere sempre maggiore del rate assegnato alla classe e deve anche essere maggiore del ceil assegnato alle sue classi figlie.

BURST

La scheda di rete può spedire solo un pacchetto alla volta e solo alla velocità legata al suo hardware. I software di link sharing possono simulare link multipli, magari con differenti velocità, ma sempre più basse di quella imposta dall'hardware utilizzato.

Per questo motivo la velocità e il limite di velocità di una classe non sono realmente istantanee, ma sono limitazioni softare che entrano in funzione dopo un certo periodo di tempo e dopo un certo numero di pacchetti, dopo i quali l'algoritmo di accodamento, adattandosi al nuovo scenario, va a impostare le varie velocità dei flussi.

Quello che accade è che il traffico da una classe viene in parte spedito alla massima velocità, dopo di che tocca alle altre classi inviare i loro pacchetti,

sempre alla massima velocità e sempre per un limitato periodo di tempo. I parametri **burst** e **cburst** controllano la quantità di dati che possono essere inviati alla massima velocità (hardware) prima che la classe sucessiva venga servita.

burst [bytes]

> Quantità di bytes che può essere inviato alla velocità massima (ceil), la quale può essere maggiore della velocità garantita (rate). Il burst di una classe dovrebbe essere almeno maggiore del burst delle sue sottoclassi.

cburst [bytes]

> Quantità di bytes che possono essere inviati alla velocità "infinita" (limite hardware), cioè alla massima velocità che l'interfaccia fisica può inviarli. Il cburst di una classe dovrebbe essere almeno maggiore del più grande cburst delle sue sottoclassi.

Quando nella limitazione della banda entra in gioco il *rate* della classe allora viene preso in considerazione il *burst*, quando invece la limitazione viene imposta dal parametro *ceil* allora si usa il *cburst*.

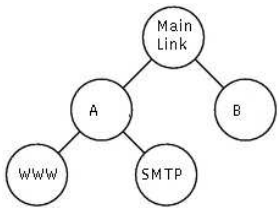
Il burst è la quantità di bytes che una classe può inviare alla velocità massima (ceil). E' importante ricordare che il limite di velocità ceil è espresso in bytes al secondo mentre il burst si misura in bytes.
Per usare una metafora potete immaginare il burst come la dimensione di un contenitore, il parametro ceil come la velocità massima con cui fuoriescono i token e il rate la velocità con la quale si riempie di token il contenitore. Così nel caso avessimo un burst da 5000 bytes, un rate di 1000 bytes al secondo e un ceil da 2500 bytes al secondo, allora potremo sostenere un traffico di 1000 bytes/sec , se in un qualsiasi istante fosse disponibile un "burst" di dati allora sarebeb possibile inviare questi ultimi alla velocità di 2500 bytes/sec, ma per quanto tempo? Bene il tempo sarà dato da 5000/2500 = 2 sec , cioè la grandezza del contenitore(burst) diviso la velocità con la quale si svuota ci da il tempo di svuotamento. Dopo di che il contenitore è di nuovo vuoto e riinizia riempirsi alla velocità di 1000 bytes/sec.

Per il cburst si applica la stessa metafora solo che adesso al posto della velocità di ceil ci starà la velocità massima che l'interfaccia hardware può sopportare.

PRIORIZZAZIONE DELLA CONDIVISIONE DI BANDA

Con il parametro *prio* è possibile impostare delle priorità con le quali al momento di dare in prestito della banda, alcune classi siano preferite alla altre.
Fino ad ora questa preferenza è stata gestita dal rapporto dei rate assegnati alle varie classi, ma se ad una classe con un rate piccolo vlessimo comunque garantire un trattamento di favore allora dovremo utilizzare le priorità esplicite.

Facendo riferimento ad un esempio di configurazione precedente:



Se assegniamo a tutte le classi la priorità 1, e solo alla classe SMTP il valore 0 (valore di priorità più elevato), allora la regola sarà che tutte le classi potranno scambiare la banda, ma alla classe con la priorità più alta verranno offerti questi eccessi prima che alle altre, ma rimangono sempre valide le regole che garantiscono la velocità (rate) e il limite massimo di velocità (ceil).

Nell'esempio precedente la cosa sarebbe evidente nel caso B cedesse parte della banda, mentre prima questa sarebbe stata distribuita 3/4 e 1/4, ora la banda va prima di tutto alla classe SMPT, se ne dovesse avanzare, magari dopo aver raggiunto il ceil della SMTP, anche a quella WWW.

CAPIRE LE STATISTICHE COL COMANDO TC

Il comando tc ci consente di ottenere delle informazioni e delle statistiche sulle discipline di accodamento sotto Linux.
Vediamo un poco la statistica sulla configurazione del paragrafo relativa alla classificazione gerarchica.
Informazioni sulle qdisc:

```
# tc -s -d qdisc show dev eth0

 qdisc pfifo 22: limit 5p
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0)

 qdisc pfifo 21: limit 5p
 Sent 2891500 bytes 5783 pkts (dropped 820, overlimits 0)

 qdisc pfifo 20: limit 5p
 Sent 1760000 bytes 3520 pkts (dropped 3320, overlimits 0)

 qdisc htb 1: r2q 10 default 1 direct_packets_stat 0
 Sent 4651500 bytes 9303 pkts (dropped 4140, overlimits 34251)
```

Le prime tre sono le qdisc figlie (PFIFO) della HTB (ultima qdisc):

- dropped ci dice quanti pacchetti sono stati scartati dalla qdisc.
- overlimits ci dice quante volte la qdisc ha dovuto ritardare un pacchetto.
- direct_packets_stat ci dice quanti pacchetti sono stati inviati direttamente attraverso la coda.

Il valore *r2q (rate to quantum)* , che di default è 10, rappresenta il fattore di conversione usato per calcolare il quantum usando la velocità della coda (rate).

Quantum = rate (in bytes)/r2q.

Informazioni sulle classi:

```
tc -s -d class show dev eth0

class htb 1:1 root prio 0 rate 800Kbit ceil 800Kbit burst 2Kb/8 mpu 0b
  cburst 2Kb/8 mpu 0b quantum 10240 level 3
  Sent 5914000 bytes 11828 pkts (dropped 0, overlimits 0)
  rate 70196bps 141pps
  lended: 6872 borrowed: 0 giànts: 0

class htb 1:2 parent 1:1 prio 0 rate 320Kbit ceil 4000Kbit burst 2Kb/8 mpu 0b
  cburst 2Kb/8 mpu 0b quantum 4096 level 2
  Sent 5914000 bytes 11828 pkts (dropped 0, overlimits 0)
  rate 70196bps 141pps
  lended: 1017 borrowed: 6872 giànts: 0

class htb 1:10 parent 1:2 leaf 20: prio 1 rate 224Kbit ceil 800Kbit burst 2Kb/8 mpu 0b
  cburst 2Kb/8 mpu 0b quantum 2867 level 0
  Sent 2269000 bytes 4538 pkts (dropped 4400, overlimits 36358)
  rate 14635bps 29pps
  lended: 2939 borrowed: 1599 giànts: 0
```

Non ho inserito le classi 1:11 e 1:12 per brevità. Come si può vedere ci sono i parametri che abbiamo impostato in fase di configurazione. Gli altri parametri riguardano i livelli e le informazioni sui quantum del DRR.
Overlimits mostra quante volte la classe ha chiesto di inviare un pacchetto ma non ha potuto per via dei limiti di banda (rate/ceil).

- rate, pps ci dice l'attuale (media su 10 sec) velocità per attraversare la classe.
- lended è il numerodi pacchetti prestato da questa classe (alla sua velocità)
- borrowed è il numero di pacchetti presi in prestito.
- giànts è il numero dei pacchetti più grandi della mtu impostata col comando tc (default 1600). HTB lavorerà con questi ma la velocità non sarà del tutto accurata.

I prestiti sono sempre calcolati in modo transitivo pe le classi, se per esempio 1:10 prendesse in prestito dei pacchetti da 1:2 e questa a sua volta li prendesse in prestito dalla 1:1, allora, anche se alla fine i pacchetti vanno dalla 1:1 alla 1:10, tutti i contatori sia delal 1:2 che della 1:10 vengono incrementati allo stesso modo.

Quantum

Il *quantum* descrive come la banda è divisa tra le varie qdisc, e funziona così:

- Tutti i quantum per tutte le qdisc sono sommate insieme e la somma viene memorizzata.
- Ogni qdisc ha una priorità relativamente alla funzione : priorità=quantum/somma.

Questa priorità viene utilizzata quando due qdiscs hanno gli stessi *rate* e *ceil*, ma gli si vuol dare una differente priorità, questa volta la priorità non riguarda il prendere o dare in prestito la banda, ma l'invio di pacchetti.

In una qdisc di tipo HTB non si può impostare direttamente come parametro, ma dalle informazione possiamo dedurre come le varie classi interagiscono in termini di priorità, riassumendo maggiore sarà il quantum della classe e maggiore sarà la sua priorità.

CLASSIFICARE I PACCHETTI CON I FILTRI

Per determinare quali classi dovranno processare un pacchetto, si usa la "catena di classificazione", qusta consiste in una serie di regole (filtri) collegata alla qdisc in cui si deve dare al scelta.

```
        root 1:
          |
         _1:1_
        /  |  \
       /   |   \
      /    |    \
    10:   11:   12:
   /   \       /   \
 10:1  10:2   12:1  12:2
```

Quando accodiamo i pacchetti, ad ogni ramo la catena dei filtri viene presa in considerazione per verificare se il pacchetto corrisponde a qualche regola.
Una tipica configurazione potrebbe essere quella con un filtro in *1:1* che dirige il pacchetto alla qdisc *12:* e un filtro sulla *12:* che lo manda alla *12:2*.

Questa ultima regola potrebbe essere inserita direttamente in *1:1* così che dalla *1:1* il pacchetto viene spedito alla *12:2*, ma distribuire il filtraggio in più stadi facilita il calcolo e rende la configurazione più leggibile.

Non si può spedire un pacchetto filtrato "*verso l'alto*", mentre, come detto prima, è possibile inviarlo verso nodi inferiori, anche saltando alcuni rami.

Un pacchetto può essere accodato solo verso il basso, mentre quando esce dalla coda ritornano di nuovo verso l'alto, dove c'è l'interfaccia (solo la qdisc root è collegata all'interfaccia).

Uso del comando tc filter

```
tc filter [ add | change | replace ] dev DEV [ parent qdisc-id | root ] protocol protocol prio priority filtertype | specific parameters ] flowid flow-id
```

```
tc filter show dev DEV
```

Il filtro deve come prima cosa avere una collocazione, e le sue coordinate sono prima di tutto l'interfaccia alla quale ci stiamo riferendo(*dev*) e successivamente l'identificativo della qdisc(*qdisc-id*) o la root alla quale lo stiamo collegando:

```
# tc filter add dev eth0 parent 10: .....
```

Dopo di che abbiamo il parametro **protocol** che altro non è che il protocollo che il classificatore accetta. Generalmente viene accettato solo il traffico IP (è obbligatorio specificarlo)

```
# tc filter add dev eth0 parent 10: protocol IP .....
```

Il parametro **prio** rappresenta la priorità di questo classificatore. Più è alto il numero e prima verrà preso in considerazione il filtro, è importante quando si hanno più liste di regole con diverse priorità.

```
# tc filter add dev eth0 parent 10: protocol IP prio 1 .....
```

A questo punto bisogna inserire il tipo di classificatore, che principalmente può essere di due tipi:

- **fw** - Che basa la sua decisione su come il firewall ha marchiato il pacchetto
- **u32** - che basa la sua decisione sulle caratteristiche del pacchetto (Ip sorgente, porta ...)
- **route** - Che basa la sua decisione sulle informazioni presenti nella tabella di routing.

IL CLASSIFICATORE "FW"

Il classificatore "fw" usa la marcatura avvenuta col firewall, per identificare e indirizzare i pacchetti. Quindi prima è necessaria l'operazione di "tagging" col firewall:

```
# iptables -I PREROUTING -t mangle -p tcp -d HostA -j MARK --set-mark 1
```

Adesso tutti i pacchetti con protocollo *tcp* che sono destinati all'host *HostA* sono contrassegnati col *mark 1*. Questi saranno riconosciuti dal filtro e saranno trattati nel metodo d'accodamento prescelto. Nel comando seguente verranno inviati alla classe *1:1* :

```
# tc filter add dev eth1 protocol ip parent 1:0 prio 1 handle 1 fw classid 1:1
```

Abbiamo collegato il filtro alla root *1:0* con priorità 1 e tutti i pacchetti col marchio 1 (handle) sono inviati alla classe *1:1*.

IL CLASSIFICATORE "U32"

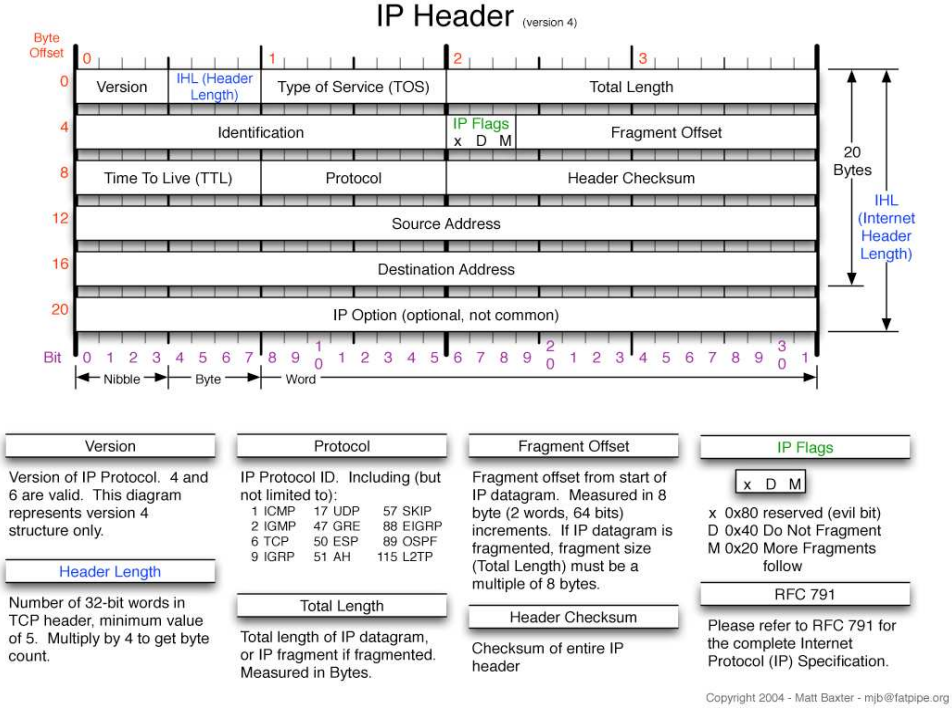
Un filtro in pratica è una lista di regole, ognuno dei quali contenente due campi: il selettore e l'azione.
Il selettore è una condizione che se viene rispettata dal pacchetto ha come conseguenza l'azione a lui associata. La lista di queste regole viene percorsa tutta fino a che non si trova un riscontro a quel punto viene eseguita l'azione e la funzione del filtro finisce. Una tipica azione potrebbe essere l'invio del pacchetto verso una classe.

Ogni lista è percorsa nell'ordine in cui sono state inserite le regole, le liste invece sono prese in considerazione in ordine di priorità dalla più alta alla più bassa (la più alta ha priorità zero).
Questa logica di liste e regole è applicabile a tutti i tipi di filtro.

Il selettore **U32** contiene la definizione di uno schema, che se trova corrispondenza nel pacchetto, lo sottopone all'azione impostata. In pratica si tratta di porre delle condizioni sui bit di intestazione di un pacchetto (header), quali indirizzo di destinazione, porta e altro ancora.

```
 # filter parent 1: protocol ip pref 10 u32 fh 800::800 order 2048 key ht 800 bkt 0 flowid 1:3 \
      match 00100000/00ff0000 at 0
```

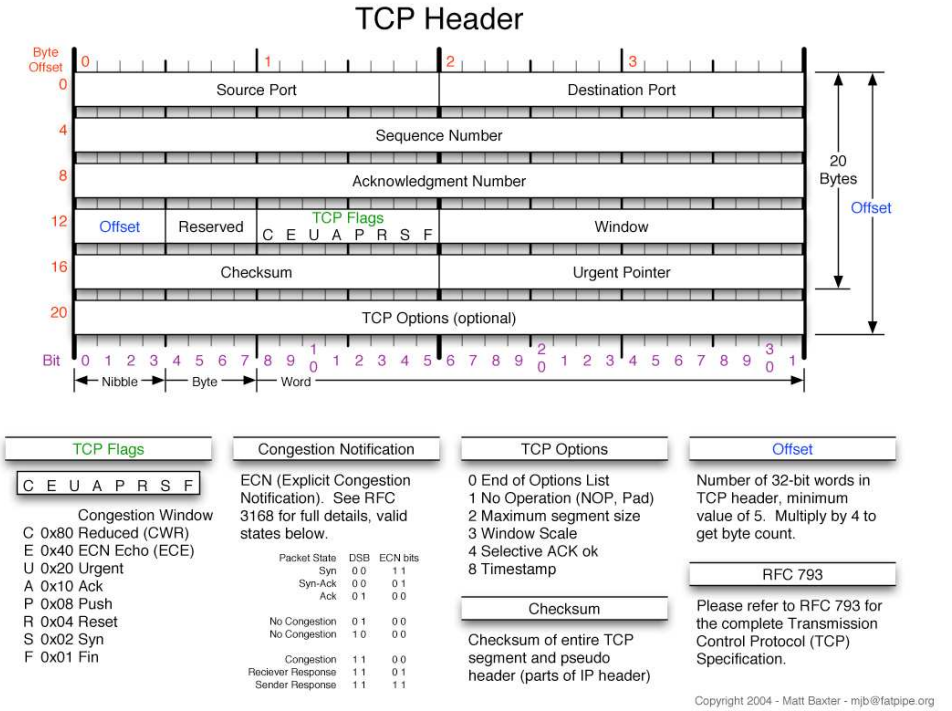
Focalizziamo l'attenzione sulla parte sottolineata dell'esempio precedente. Essa contiene la condizione di match, tramite questa il selettore cercherà tutti i pacchetti, il cui IP header abbia il secondo byte paria a *10*. Come potete immaginare 00ff0000 è la maschera di match che fa si che si prenda in considerazione solo il secondo byte dell'header del pacchetto, mentre i restanti bit non vengono presi in considerazione. La parola *at* invece indica l'offset (in bytes) dall'inizio del pacchetto. Se guardiamo a cosa corrispondono questi bit nell'immagine qua sotto capiamo che abbiamo messo una condizione suo TOS (Type of Service), in particolare sul campo che identifica i pacchetti con un basso ritardo.



Analizziamo ques'altra regola:

```
# filter parent 1: protocol ip pref 10 u32 fh 800::803 order 2051 key ht 800 bkt 0 \
    flowid 1:3 \
    match 00000016/0000ffff at nexthdr+0
```

L'opzione *nexthdr* significa "il prossimo header incapsulato nel pacchetto IP", per esempio potrebbe essere l'header TCP e UDP, dove il terzo e quarto byte (la maschera è *00 00 FF FF*) corrispondono alla porta di destinazione. La condizione sarà soddisfatta se come porta di destinazione si avrà il valore *0x0016* che tradotto da esadecimale in decimale corrisponde al numero 22.
In pratica abbiamo a che fare con un filtro che seleziona i pacchetti destinati alla porta 22 cioè una connessione SSH, sempre che si tratti di una connessione TCP.



Avendo capito tutto quello spiegato qua sopra, sarà facile capire questa condizione di match:

```
match c0a80100/ffffff00 at 16
```

I primi 3 byte dal 17°byte del IP header (destination address) devono essere pari a "c0 a8 01" che tradotto in decimali significa "192 168 1", in pratica seleziona tutti i pacchetti destinati agli indirizzi di rete di tipo 192.168.1/24.

Selettori generali

I *general selectors* definiscono uno schema, maschera e offset che permette di selezionare il contenuto dei pacchetti, infatti è possibile creare delle regole

per ogni singolo bit all'interno dell'header IP o degli heder superiori (incapsulati). La sintassi descritta qui sotto è semplice ma un po difficile da leggere:

```
match [ u32 | u16 | u8 ] PATTERN MASK [ at OFFSET | nexthdr+OFFSET]
```

Le parole chiave *u32*, *u16* o *u8* specificano la lunghezza del pattern in bit. Il PATTERN e la maschera MASK devono avere la stessa lunghezza in bit della parola chiave precedente. L' OFFSET è il punto, rispetto all'inizio dell'header, da cui parte il confronto (in bytes). Se davanti al numero di offset è presente la parola chiave *nexthdr* , allora l'offeset sarà il numero di bytes dall'inizio dell'header successivo.
Qualche esempio:

```
    # tc filter add dev ppp14 parent 1:0 prio 10 u32 \
         match u8 64 0xff at 8 flowid 1:4
```

Il pacchetto soddisferà la condizione all'interno della regola, se il suo time to live (TTL) è pari a 64. Il TTL è il campo all'interno dell'header partendo dal ottavo byte.

```
    # tc filter add dev ppp14 parent 1:0 prio 10 u32 \
         match u8 0x10 0xff at nexthdr+13 \
         protocol tcp flowid 1:3 \
```

Questa regola riconoscerà solo il traffico TCP con il bit di ACK impostato ad 1.
Se diamo un'occhiata al diagramma dell'header TCP possiamo vedere il bit di ACK è il secondo bit (0x10) nel 14°bute del header TCP (*at nexthdr+13*).
Come secondo selettore al posto di "*protocol TCP*" potremo scrivere "match u8 0x06 0xff at 9", dato che 6 è il codice del protocollo TCP presente nel 10° byte dell'header IP.

All'interno del selettore u32 possono essere inclusi molti match, così da comporre un filtro più complesso:

```
# tc filter add dev eth0 parent 1:0 prio 1 u32 \
         match ..... \
         match ..... \
         match ..... \
         flowid 1:2 \
```

Le varie condizioni all'interno del selettore è come se fossero unite dall'operatore AND, per cui la selezione avrà esito positivo solo se tutte le condizioni verranno rispettate.

Selettori specifici

I selettori specifici ci permettono di accedere ai principali bit dell'header in maniera più semplice, quinid useremo:

- match ip protocol *n* 0xff Con cui ci riferiremo direttamente al campo che descrive il tipo di protocollo ip nell'header, ad esempio se *n* avrà il valore *6* selezioneremo i pacchetti con protocollo tcp.
- match ip sport *n* 0xffff Con questo ci riferiremo alla porta sorgente del protocollo tcp o udp.
- match ip dport *n* 0xffff Con questo ci riferiremo alla porta destinazione del protocollo tcp o udp.
- match ip src *1.2.3.4/24* Con questo ci riferiremo ad un indirizzo o una classe di indirizzi come sorgente.
- match ip dst *1.2.3.4/24* Con questo ci riferiremo ad un indirizzo o una classe di indirizzi come destinazione.
- match ip tos *0xn* 0xff Con questo ci riferiremo al campo tos del pacchetto.

in questo modo le regole alla base della classificazione saranno molto più leggibili.

IL CLASSIFICATORE "ROUTE"

Questo filtro classificatore è basato sui risultati delle decisioni della tabella di instradamento (routing table). Quando un pacchetto, che sta attraversando il nodo incontra una regola con il filtro di tipo "route", viene classificato a seconda delle infomarioni fornite dal precedente instradamento nella tabella di route.

```
    # tc filter add dev eth1 parent 1:0 protocol ip prio 100 route to 10 classid 1:10
```

Con il precedente comando abbiamo collegato al nodo 1:0 (root qdisc) con priorità 100 il filtro di tipo *route*. Quando il pacchetto raggiunge questo nodo (in questo caso trattandosi del nodo root, cioè il primo, questo accade subito) il filtro andrà a controllare la tabella di route per trovare l'instradamento ricevuto dal pacchetto e a seconda della destinazione viene classificato nella classe prescelta.
Per fare ciò la tabella di routing deve essere definita nel modo apropriato, cioè deve essere definito il "realm" di una route:

```
# ip route add Host/Network via Gateway dev Device realm RealmNumber
```

Per esempio, possiamo definire una route che tutti i pacchetti provenienti dalla classe IP *192.168.10.0/24* siano instradati verso la destinazione di rete *192.168.10.0* marcata con il *realm number 10*:

```
# ip route add 192.168.10.0/24 via 192.168.10.1 dev eth1 realm 10
```

Aggiungendo il seguente filtro *route*, noi possiamo usare il numero realm per riconoscere i pacchetti che sono stati instradati con la precedente regola di instradamento (route).

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 \
  route to 10 classid 1:10
```

Una volta che sono stati identificati saranno inviati alla classe *1:10*.

I filtri route possono anche essere usati per selezionare il traffico in base alle source routes, cioè agli indirizzi ip sorgenti nella regola di instradamento. Per esempio, qui una sottorete è colelgata al router linux sulla interfaccia di rete eth2.

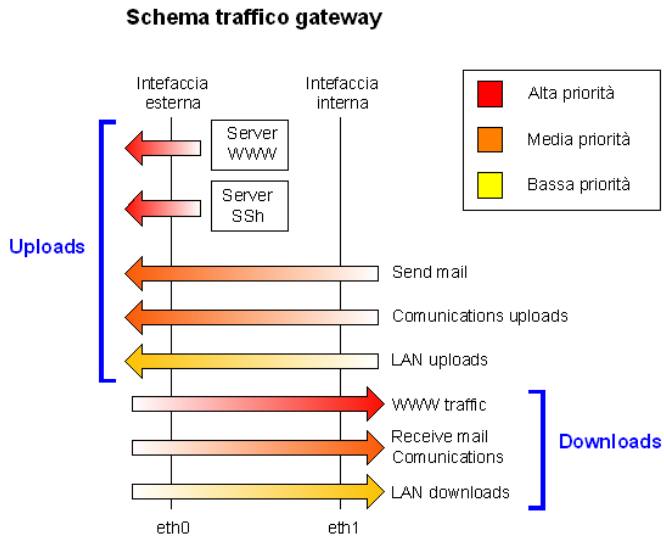
```
# ip route add 192.168.2.0/24 dev eth2 realm 2
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 \
  route from 2 classid 1:2
```

In questo modo il filtro manda i pacchetti provenienti dalla sottorete *192.168.2.0* (realm 2) nella classe con identificativo *1:2*.
Per maggiori informazioni riguardo le tecniche di instradamento dei pacchetti leggere l'articolo "Guida la packet routing".

REGOLAZIONE DEL TRAFFICO SU UN GATEWAY

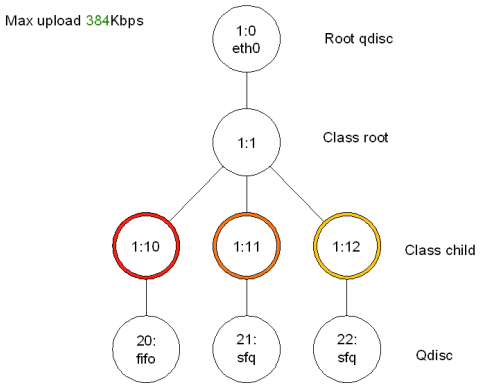
Per spiegare meglio i concetti fin qui esposti facciamo un esempio pratico dell'uso degli accodamenti, delle classi e dei filtri.

La situazione in cui ci troviamo è la seguente:
Siamo connessi ad internet tramite una ADSL che ci viene garantita per *7Mbps* in download e *384Kbps* in upload. Direttamente connesso alla adsl abbiamo messo un pc, il quale fungerà da gateway. Questo avrà due interfacce di rete, una con la quale si connette ad internet (esterno) chiamata *eth0*, e un altra (*eth1*) che lo connette alla lan interna. In questo modo il pc in questione metterà in comunicazione i pc della rete interna con l'esterno, proteggendoli e regolandone il traffico. Un altra caratteristica è che il mio gateway avrà al suo interno anche due server (Webserver e SSH server) che dovranno essere accessibili dall'esterno sulla porta 80 e sulla porta 22.



Ora, come mostra la figura qui sopra, ci sono in gioco più tipi di traffico (www,mail,...) aventi due direzioni diverse (upload/download) e distribuiti sulle 2 interfacce (eth0/eth1). Noi dovremo garantire con il controllo del traffico che ad esempio le persone, che visitano il sito web ospitato nel gateway, abbiano la priorità sugli upload della rete interna (invio di mail, upload di file, etc...), e che gli utenti della rete interna possano navigare in internet (www) senza che vengano disturbati dagli altri tipi di traffico.
Questo è in breve il nostro fine, ma l'immagine descrive meglio quello che andremo a realizzare.
Innanzitutto abbiamo 2 interfacce diverse, quindi avremo due schemi gerarchici separati, uno per la eth0 l'altro per la eth1.

Ora andiamo avanti affrontando un altro importante concetto: le code che andramo ad impostare sono del tipo *egress* quindi sono code in uscita. Questo significa che del traffico che passa ad esempio nella interfaccia eth0 io andro ad occuparmi solo di quello uscente ovvero degli uploads, il restante traffico non viene regolato su questa interfaccia, ma la sua gestione avviane nella interfaccia interna eth1.
Allo stesso modo del traffico che passa per l'interfaccia interna andremo a comprollare solo quello che definiremo come download.



Nella immagine qui sopra è mostrata la gerarchia di qdisc e classi che andremo ad usare.
Innanzi tutto verrà creata la qdisc di root *1:* , collegata direttamente all'interfaccia eth0.

```
tc qdisc add dev eth0 root handle 1: htb default 12
```

Notare che di default, cioè se non vengono classificati da nessun filtro, i pacchetti verranno inviati alla classe *1:12*.

Dopo verrà creata la class root *1:1* alla quale collego le 3 classi figlie *1:10*, *1:11*, *1:12*, ognuna con un proprio rate impostato a seconda del tipo di traffico di cui dovranno occuparsi.

```
tc class add dev eth0 parent 1: classid 1:1 htb rate 384Kbps
tc class add dev eth0 parent 1:1 classid 1:10 htb rate 200kbps ceil 384Kbps
tc class add dev eth0 parent 1:1 classid 1:11 htb rate 180kbps ceil 384Kbps
tc class add dev eth0 parent 1:1 classid 1:12 htb rate 20kbps ceil 384Kbps
```

Il traffico verrà quindi smistato in tre classi diverse a seconda della priorità che gli verrà assegnata. Nello schema precedente abbiamo identificato gli uploads a seconda della loro natura in tre tipi di priorità. In particolare il traffico proveniente dai due server (www ed ssh) del gateway dovranno avere priorità alta e saranno inviati alla classe 1:10, le email in uscita e le comunicazioni iterative (video chiamate, etc..) avrano priorità media e saranno inviate verso la classe 1:11, tutto il restante traffico in uscita (default) sarà inviato alla classe 1:12.

Vediamo un esempio di come verrano impostati i filtri, collegandoli direttamente alla root qdisc:

```
tc filter add dev $EXT parent 1: protocol ip u32 match ip sport 80 0xffff flowid 1:10
tc filter add dev $EXT parent 1: protocol ip u32 match ip sport 222 0xffff flowid 1:10
tc filter add dev $EXT parent 1: protocol ip u32 match ip dport 80 0xffff flowid 1:10
tc filter add dev $EXT parent 1: protocol ip u32 match ip dport 53 0xffff flowid 1:10
tc filter add dev $EXT parent 1: protocol ip u32 match ip dport 25 0xffff flowid 1:11
tc filter add dev $EXT parent 1: protocol ip u32 match ip dport 110 0xffff flowid 1:11
```

I primi due filtri riguardano il traffico proveniente dal gateway stesso, ed hanno infatti come parametro di riferimento la porta sorgente (sport), mentre gli altri quattro riguardano l'invio delle mail e le richieste dns e www dalla lan e la porta questa volta sarà quella di destinazione (dport). Tutti gli altri pacchetti che non corrispondono a queste specifiche saranno inviati alla classe *1:12*.

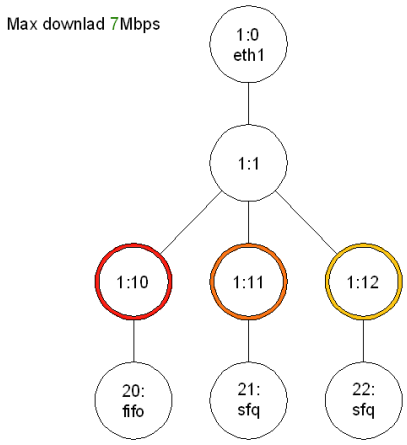
Ogni classe avrà una banda garantita, ma in caso fosse possibile potranno scambiarsi la banda tra loro, questo grazie al fatto che le tre classi figlie non sono collegate direttamente alla qdisc di root ma ad un altra classe (1:1), la quale garantisce la condivisione e lo scambio di banda tra le sue discendenti.

Sotto le classi figlie vi sono le qdisc finali, che di solito sono di tipo *pfifo*. Quindi andremo a impostare esplicitamente solo quelle che modificheremo in *sfq*.

```
tc qdisc add dev eth0 parent 1:11 handle 21: sfq
tc qdisc add dev eth0 parent 1:12 handle 22: sfq
```

La vera e propria azione di classificazione la otterremo con i filtri, i quali saranno collegati direttamente alla qdisc root, e smisteranno il traffico nelle varie classi figlie.

Lo stesso discorso vale per la interfaccia interna eth1:



Dove però questa volta il limite massimo è di *7Mbps*, quindi come *ceil* dovremo impostare questo valore.

```
tc qdisc del dev eth1 root
tc qdisc add dev eth1 root handle 1: htb default 12
tc class add dev eth1 parent 1: classid 1:1 htb rate 7Mbps
tc class add dev eth1 parent 1:1 classid 1:10 htb rate 200kbps ceil 7Mbps
tc class add dev eth1 parent 1:1 classid 1:11 htb rate 180kbps ceil 7Mbps
tc class add dev eth1 parent 1:1 classid 1:12 htb rate 20kbps ceil 7Mbps
tc filter add dev eth1 parent 1: protocol ip u32 match ip sport 80 0xffff flowid 1:10
tc filter add dev eth1 parent 1: protocol ip u32 match ip sport 53 0xffff flowid 1:10
tc filter add dev eth1 parent 1: protocol ip u32 match ip sport 25 0xffff flowid 1:11
tc filter add dev eth1 parent 1: protocol ip u32 match ip sport 110 0xffff flowid 1:11
tc qdisc add dev eth1 parent 1:11 handle 21: sfq
tc qdisc add dev eth1 parent 1:12 handle 22: sfq
```

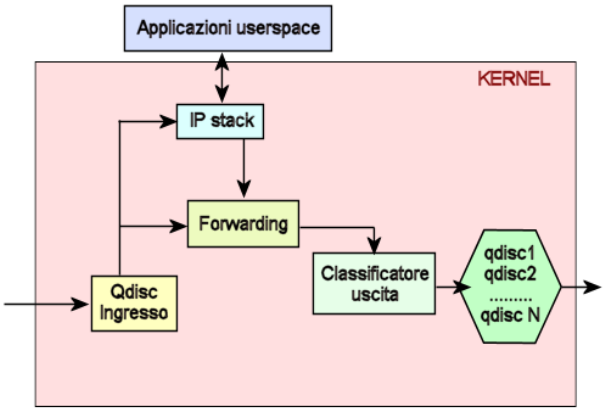
Il filtraggio questa volta è rivolto a catturare il traffico dall'esterno verso l'interno della lan quindi le porte filtrate saranno quelle sorgenti sport, in maniera complementare a quelle di destinazione usate nella interfaccia eth0.

Da notare che non abbiamo previsto una classificazione ad alta priorità per il traffico tra la lan e i server sul gateway, ma questo non vuol dire che non vi si potrà accedere.

Tengo a precisare che questo è solo un esempio e che quindi serve solo a spiegare il metodo e il ragionamento per configurare gli accodamenti, poi le situazioni reali sono molto più complesse e necessitano di uno studio preliminare del traffico in gioco.

INGRESS QDISC

Le qdisc di cui abbiamo parlato fino ad ora sono *egress qdisc*, cioè stanno all'uscita dell'interfaccia e si occupano di gestirne i flussi in uscita. Ogni interfaccia ha comunque una qdisc di ingresso, la quale si occupa di applicare i *tc filters* ai pacchetti in ingresso, indipendentemente dalla loro destinazione o da che uso se ne dovrà fare.



La sua implementazione garantisce molte delle funzionalità che abbiamo visto per le code ed i filtri in uscita, grazie a queste sarà possibile eseguire le operazioni di policing al traffico in arrivo, prima che entri nell'IP stack e venga ulterirmente elaborato. Questo ci permette di risparmiare molta della potenza di calcolo che normalmente viene utilizzata per le operazioni di accodamento e classificazione.

La ingress qdisc non richiede molti parametri e differisce dalle altre per il fatto che non va ad occupare la root dell'interfaccia:

```
# tc qdisc add dev eth0 ingress
```

Dopo di che dovremo collegarci i filtri di policing che solitamente come azione hanno il *drop*, cioè scartano i pacchetti che non corrispondono alla regola del filtro.

Le altre azioni disponibili sono:

- continue: che cause una condizione di non-match, che magari si avrà dopo con altri filtri.
- Pass/OK: che fa passare il traffico ok. Solitamente viene utilizzata per disabilitare un filtro, senza cancellarlo.
- reclassify: Fa si che avvenga di nuovo la riclassificazioen per un Best Effort.Questa è l'azione di default se non ne viene specificata una.

Ricordo alcuni comandi utili:

Per elencare le qdisc su una interfaccia

```
tc qdisc ls dev eth0
```

Per elencare le classi su una interfaccia

```
tc class ls dev eth0
```

Per elencare i filtri su una interfaccia

```
tc filter ls dev eth0 parent ffff:
```

Per cancellare la qdisc di ingresso

```
tc qdisc del eth0 ingress
```

Vediamo un esempio:

Proteggere un pc dal SYN flood

Quello che andremo a fare è impostare una qdisc all'ingresso dell'interfaccia che attraverso un filtro ci protegga contro un eccessivo bombardamento di pacchetti SYN. Non voglio usare il firewall dato che con esso sarei obbligato a imporre una regola ben precisa: o il pacchetto passa oppure viene scartato. Noi invece vogliamo limitare l'eccesso di pacchetti non eliminarli tutti.
Come prima cosa imposto una regola del firewall che marchi ogni pacchetto SYN in arrivo sulla interfaccia eth0.

```
$iptables -A PREROUTING -i eth0 -t mangle -p tcp --syn \
  -j MARK --set-mark 1
```

Installo la qdisc in ingresso nell'interfaccia eth0 chiamandola *"ffff:"*, questo identificativo è la qdisc in ingress stessa, tant'è che potevano anche ometterlo:

```
tc qdisc add dev eth0 handle ffff: ingress
```

Ora facciamo un piccolo calcolo:
i pacchetti SYN sono di 40 bytes (320 bits) così 3 di essi sono circa 960 bits (circa 1kbit); così noi imponiamo un limite ai SYN in arrivo di 3 pacchetti al secondo, naturamente questo è solo un esempio banale poi siete liberi di imporre le soglie che volete.

```
tc filter add dev eth0 parent ffff: protocol ip prio 50 handle 1 fw \
   police rate 1kbit burst 40 mtu 9k drop flowid :1
```

in questo modo se il flusso di SYN non rispetta i limiti imposti, nel nostro caso è di 1 Kbps , verrà scartato (drop) altrimenti continua.

Qualche altro esempio di filtri di policing:

I seguente filtro limita il traffico icmp in ingresso a 2kbit, scartando i pacchetti se viene superata questa soglia:

```
tc filter add dev $DEV parent ffff: \
    protocol ip prio 20 \
    u32 match ip protocol 1 0xff \
    police rate 2kbit buffer 10k drop \
    flowid :1
```

Scarta i pacchetti di lunghezza superiore a 84 bytes:

```
tc filter add dev $DEV parent ffff: \
   protocol ip prio 20 \
   u32 match tos 0 0 \
   police mtu 84 drop \
   flowid :1
```

Scarta tutti i pacchetti icmp:

```
tc filter add dev $DEV parent ffff: \
   protocol ip prio 20 \
   u32 match ip protocol 1 0xff \
   police mtu 1 drop \
   flowid :1
```

In realtà scarta tutti i pacchetti superiori a 1 byte, quindi quelli di 1 byte potrabbero passare, ma nella realtà non li troverete mai.

Filtra "tutto" (0.0.0.0/0), scartando tutto quello che arriva troppo velocemente.

```
tc filter add dev eth0 parent ffff: protocol ip prio 50 u32 match ip src \
   0.0.0.0/0 police rate 7Mbps burst 10k drop flowid :1
```

In questo caso la velocità massima permessa sarà di 7Mbps.

INTERMEDIATE QUEUEING DEVICE (IMQ)

Fino ad ora abbiamo detto che non è possibile avere lo shaping del traffico all'ingresso di una interfaccia, ma ricorrendo alla IMQ ovvero Intermediate Quequing Device possiamo fare in modo di applicare tutte le tecniche di accodamento viste finora anche agli ingressi delle interfacce.

Di default solitamente dovremo avere due dispositivi imq (imq0 e imq1). Questi rappresentano delle interfacce fittizie, con le quali non si può fare nullaltro che collegarci delle qdisc e quindi applicarci tutte le tecniche viste fino ad ora.

Per prima cosa bisogna quindi collegare una qdisc al dispositivo imq, questo sarà trattato come un altra interfaccia di rete.

Dopo di che bisogna specificare quali pacchetti dovranno passare attraveso questo dispositivo. A questo scopo noi useremo le iptables con target (`IMQ') per selezionare e inviare i pacchetti verso la nostra interaccia virtuale.

AppenaAs l'interfaccia raggiunge lo stato IFF_UP i pacchetti incominceranno ad essere accodati nelle sue qdisc. Quando arriverà il momento di far uscire i pacchetti da questa interfaccia virtuale e quindi quando saranno estratti dalle code saranno inviati nello stack di rete.

Per verificare se sono installati e funzionanti i dispositivi IMQ bisogna prima verificare se sono stati installati come moduli del kernel ed eventualmente caricarli con i comandi "*modprobe imq*" e "*modprobe ipt_IMQ*". Se fossero di più di quelli previsti in fase di compilazione bisogna specificarne il numero: "*modprobe imq numdevs=X*", dove *numdev* è il numero dei dispositivi in questione.
Se scoprite che non avete installato IMQ per il kernel che state usando, allora non potrete usare questi comandi fino a che non avrete risolto il problema, che in molti casi si traduce nella ricompilazione del kernel stesso.

Facciamo un esempio di configurazione per imq:

```
#collego la qdisc di base all'interfaccia imq
tc qdisc add dev imq0 root handle 1: htb default 20
#vi collego la classe root con un limite di banda a 2 mega
tc class add dev imq0 parent 1: classid 1:1 htb rate 2mbit burst 15k
#creo 2 classi figlie assegnando a ciascuna 1 mega
tc class add dev imq0 parent 1:1 classid 1:10 htb rate 1mbit
tc class add dev imq0 parent 1:1 classid 1:20 htb rate 1mbit
#imposto le qdisc sulle classi figlie
tc qdisc add dev imq0 parent 1:10 handle 10: pfifo
tc qdisc add dev imq0 parent 1:20 handle 20: sfq
#creo un filtro che invia i pacchetti con destinazione 10.0.0.230/32
#al nodo 1:10, mentre tutti gli altri di default vanno alla 1:20
tc filter add dev imq0 parent 1: protocol ip prio 1 u32 match \
        ip dst 10.0.0.230/32 flowid 1:10
```

Questa configurazione è del tutto analoga a quelle sulle interfacce reali che abbiamo già visto, ma per funzionare correttamente necessita di ulteriori comandi:

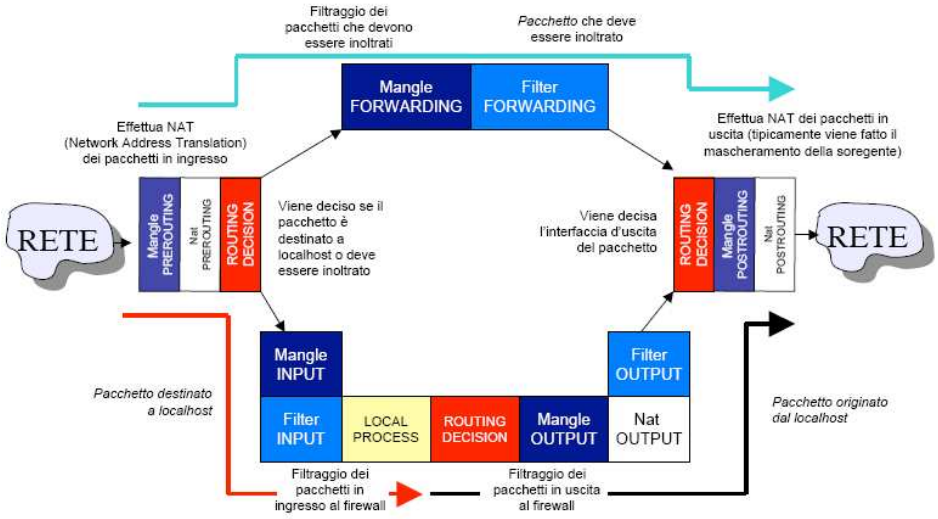
```
iptables -t mangle -A PREROUTING -i eth0 -j IMQ --todev 0
ip link set imq0 up
```

Col primo comando impongo che tutti i pacchetti in ingresso nell'interfaccia *eth0* vengano inviari alla IMQ, mentre col secondo attivo la IMQ.

Nelle iptables il target di tipo *IMQ* è valido solo nelle catene di PREROUTING e POSTROUTING della tabella mangle. La sua sintassi è:

```
IMQ [ --todev n ]    n : numero della interfaccia imq
```

Come è mostrato in questo grafico



la tabella a cui ci siamo riferiti è proprio all'ingresso dell'interfaccia, mentre quella di postrouting e appena prima dell'uscita, ad ogni modo la sintassi per deviare i pacchetti di pre o post routing è la seguente:

Per i pacchetti in ingresso:

```
iptables -t mangle -A PREROUTING [condizioni] -j IMQ --todev 0  # questi pacchetti vanno alla imq0
iptables -t mangle -A PREROUTING [condizioni] -j IMQ --todev 1  # questi pacchetti vanno alla imq1
iptables -t mangle -A PREROUTING [condizioni] -j IMQ --todev 2  # questi pacchetti vanno alla imq2
...
```

Per i pacchetti in uscita:

```
iptables -t mangle -A POSTROUTING [condizioni] -j IMQ --todev 0  # questi pacchetti vanno alla imq0
iptables -t mangle -A POSTROUTING [condizioni] -j IMQ --todev 1  # questi pacchetti vanno alla imq1
iptables -t mangle -A POSTROUTING [condizioni] -j IMQ --todev 2  # questi pacchetti vanno alla imq2
...
```

Ultimo aggiornamento (Venerdì 18 Giugno 2010 20:06)