

Sterowanie przepływem danych w Linuxie 2.2

Paweł Krawczyk
kravietz@ceti.pl

14 sierpnia 2001

Spis treści

1	Wprowadzenie	3
2	Droga pakietu przez ruter	4
3	Algorytmy kolejkowania	5
3.1	Prosta kolejka	5
3.2	Kolejka złożona	5
4	Filtry	6
4.1	route	6
4.2	fw	6
4.3	u32	7
5	Elementarne algorytmy kolejkowania	7
5.1	FIFO	7
5.2	TBF	7
5.3	PRIOR	8
5.4	SFQ	8
5.5	RED	9
5.6	CBQ	10
6	Rozdzielczość zegara	12
7	Proste przykłady zastosowania kolejek	13
7.1	Domyślne ustawienia	13
7.2	Prosta kolejka PRIOR	13
7.3	Ograniczenie pasma (TBF)	14
7.4	Automatyczne współdzielenie łącza (SFQ)	14

8	Podział łącza na klasy za pomocą CBQ	15
8.1	Klasyfikacja filtrem fw	17
8.2	Klasyfikacja filtrem route	18
8.3	Klasyfikacja filtrem u32	18
9	Dodatki	19
9.1	Jednostki przepustowości łącz	19
10	Filtr u32	20
10.1	Selektor u32	21
10.2	Parametry ogólne	22
10.3	Parametry szczegółowe	22
11	Inne implementacje	24

1 Wprowadzenie

Sterowanie przepływem danych (*traffic control*) jest kolejną nowością w Linuxie 2.2. Właściwie już w kernelach 2.1 zaimplementowano prosty *traffic shaper*, ale posiadał on dość poważne ograniczenia i służył wyłącznie do ograczania ruchu (stąd nazwa *shaping*). Od tej pory Linux zyskał bardzo bogaty zespół procedur służących nie tylko do ograniczania ruchu, ale – ogólniej – do sterowania przepływem danych. Ich zastosowanie to przede wszystkim optymalizacja wykorzystania łącz, oraz zapewnianie określonych parametrów łącza dla wybranych transmisji.

Wszystko to wiąże się z bardzo obszernymi i szeroko obecnie dyskutowanymi zagadnieniami usług o gwarantowanej jakości (*quality of service*), o które będziemy się nieustannie ocierać podczas omawiania tego co Linux ma do zaoferowania w tej dziedzinie.

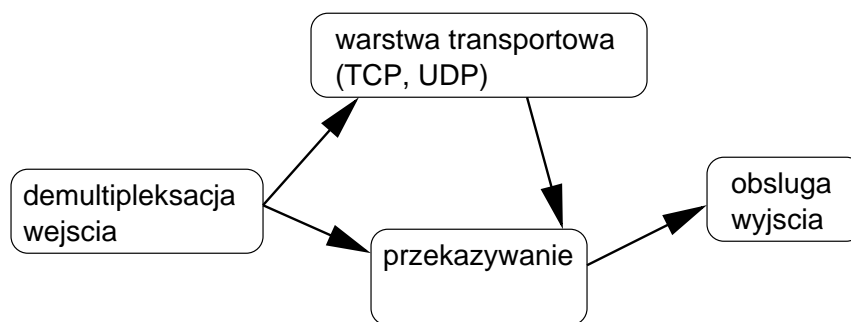
Na początek musimy poczynić kilka założeń. Po pierwsze, łącza mają ograniczoną przepustowość. Po drugie, przepustowości łącz w Internecie są bardzo zróżnicowane. Router może przyjmować dane z Ethernetu z prędkościami rzędu megabitów na sekundę i wysyłać je łączem szeregowym o przepustowości o dwa rzędy wielkości mniejszej. Większość omawianych poniżej procesów zachodzi na styku takiego szybkiego i wolnego łącza, co siłą rzeczy prowadzi do sytuacji, gdy część pakietów nie może zostać „wepchnięta” w wąskie gardło i jest gubiona.

Zastosowanie w tym miejscu sterowanie przepływem danych pozwala ograniczyć straty do minimum, poprawić wykorzystanie łącza, a także – w razie potrzeby – zapewnić określonym rodzajom danych pierwszeństwo.

I jeszcze uwaga na temat samego dokumentu. Stanowi on drugi z cyklu artykułów na temat nowości w zakresie obsługi protokołu IP, jakie pojawiły się w Linuxie 2.2. Jest przeznaczony dla średnio zaawansowanych administratorów sieci, chcących lepiej wykorzystać dostępne im zasoby. Funkcjonalnie składa się z dwóch części – teoretycznej (rozdziały 2–6) i praktycznej (rozdział 7). W pierwszej starałem się przybliżyć podstawy sterowania przepływem danych w sieciach IP, ukierunkowane na implementację linuxową. Druga część służy jako podręczne kompendium do konfiguracji QoS w działających sieciach.

2 Droga pakietu przez ruter

Poniższy schemat przedstawia działanie interesującego nas fragmentu funkcjonalnego rutera. Dane przychodzą z kilku interfejsów i są przez ruter demultipleksowane. **Demultipleksacja** obejmuje usunięcie nagłówków warstwy łącza oraz przyporządkowanie pakietu do danego protokołu warstwy sieciowej, a następnie transportowej – jeśli jest on skierowany do tego hosta.



Rys. 1: Droga pakietu przez ruter.

W przypadku rutera większość ruchu stanowią pakiety kierowane do innych hostów. Ich obsługa to część schematu opisana jako **przekazywanie** (*forwarding*) – obejmuje to przede wszystkim wyznaczenie adresu następnego rutera oraz interfejsu docelowego dla takiego pakietu.

Dane przeznaczone do wysłania przez określony interfejs trafiają do kolejki wyjściowej (**obsługa wyjścia**), z której są usuwane tak szybko jak to jest stanie zrealizować dany interfejs.

Jak wspomnieliśmy wyżej, interesują nas głównie przypadki gdy danych w kolejce wyjściowej przybywa szybciej, niż interfejs jest w stanie je z niej pobierać i wysyłać do celu. Jeśli kolejka ma wystarczająco dużą pojemność, a przeciążenie nie trwa zbyt długo, to istnieje możliwość że pakiety zostaną stopniowo wysłane przez interfejs. W przeciwnym razie część z nich zostanie zgubiona. O tym, które pakiety zostaną zgubione decyduje algorytm obowiązujący w danym buforze-kolejce.

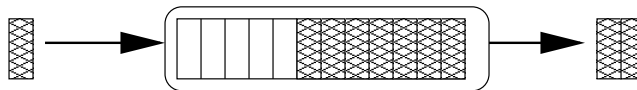
Istotną konsekwencją takiego modelu jest to, że sterowanie przepustowością dotyczy wyłącznie ruchu **wychodzącego** z rutera. W kontekście wykorzystania łącz jest to jednak całkowicie uzasadnione i wystarczające – ruter otrzymujący dane z przeciążonego łącza nie może za pomocą kolejkowania zmniejszyć przeciążenia – może to zrobić wyłącznie ruter po drugiej stronie, będący źródłem danych. Ograniczenie ruchu przychodzącego do wybranej grupy hostów w celach administracyjnych można natomiast osiągnąć regulując przydzielone tej grupie pasmo na interfejsie ich rutera.

3 Algorytmy kolejowania

W nomenklaturze angielskiej określany jako *queueing discipline* termin ten ma trochę szersze znaczenie niż wynika z polskiego tłumaczenia. Ponieważ określenie „algorytm kolejowania” jest dość nieporęczne, będziemy go poniżej używać zamiennie z określeniem „kolejka”. Ogólnie rzecz biorąc, algorytm kolejowania decyduje w jakiej kolejności przeznaczone są do wysłania pakiety znajdujące się aktualnie w kolejce. Istotne jest, że może on być albo jednym z elementarnych algorytmów opisanych poniżej, albo stanowić złożoną strukturę, podzieloną na klasy z przyporządkowanymi wieloma algorytmami elementarnymi. Poniższe schematy ilustrują oba te przypadki.

3.1 Prosta kolejka

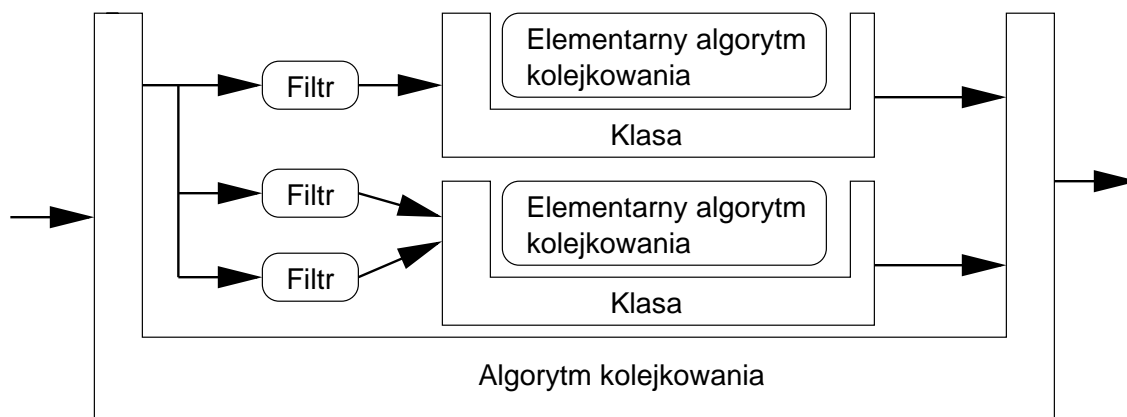
Prosta kolejka z rysunku to zazwyczaj FIFO (patrz niżej). Taki algorytm jest używany domyślnie na interfejsach Ethernet, PPP i innych, obsługiwanych przez Linuxa. Równie dobrze może to być jednak jeden z algorytmów opisanych poniżej – TBF, SFQ itp.



Rys. 2: Prosta kolejka.

3.2 Kolejka złożona

Kolejka złożona może być bardzo rozbudowaną strukturą, zawierającą w sobie kolejki proste oraz inne elementy, takie jak filtry i klasy.



Rys. 3: Złożona kolejka.

Składniki kolejki złożonej:

- **Filtry**, odpowiadające za przyporządkowanie pakietów do odpowiednich klas na podstawie wybranych parametrów, takich jak adres źródłowy, docelowy, protokół i wiele innych.
- **Klasy**, posiadające różne priorytety i stanowiące właściwe rozróżnienie między różnymi rodzajami ruchu.
- **Elementarne algorytmy kolejkowania**, decydujące o sposobie obsługi pakietów, które trafiły do danej klasy.

4 Filtry

Pakiet wchodzący do złożonej kolejki jest klasyfikowany do określonej klasy przez filtr, kierujący się wybranymi parametrami pakietu. W zależności od rodzaju filtra mogą to być: adres źródłowy i docelowy pakietu, port, protokół, TOS itp. W chwili obecnej w linuxowej implementacji QoS dostępne są trzy podstawowe filtry – **route**, **fw** oraz **u32**.

4.1 route

Filtr oparty o tablice routingu. Każda trasa w tablicy routingu może mieć przypisane oznaczenie kolejki, do której mają być kierowane pakiety kierowane według tej trasy.

Filtr **route** pozwala na klasyfikację pakietów ze względu na te same parametry, które są używane podczas wybierania dla niego trasy w tablicy routingu – a więc adresu docelowego (lub źródłowego, przy zastosowaniu routingu rozszerzonego). Jego największą zaletą jest szybkość oraz mały narzut czasowy podczas klasyfikacji, wynikające z dużej efektywności operacji na tablicy routingu.

4.2 fw

Filtr **firewall**. Opiera się o zaznaczanie pakietów przez firewall wbudowany w kernel. W przypadku *ipchains* do konfiguracji regułki filtra należy dodać opcję **-m**, której parametrem jest liczba stanowiąca oznaczenie pakietu. Jeśli przetwarzany przez firewall pakiet pasuje do danej regułki, to zostaje on oznaczony podaną liczbą. Faktycznie oznaczenie polega na ustawieniu pola **skb->priority** w pakiecie na podaną wartość. Warto zaznaczyć, że pole to jest także ustawiane przez jądro w zależności od TOS pakietu.

Istotna jest interpretacja oznaczenia. Jest ono liczbą 32-bitową, której starsze 16 bitów określa kolejkę do której ma być skierowany pakiet, a młodsze – klasę w obrębie tej kolejki. Przykładowo, chcąc skierować pakiet do klasy 1:3 powinniśmy użyć

parametru `-m 0x10003`. Filtr `fw` pozwala na klasyfikację pakietów według wszystkich parametrów rozpoznawanych przez firewall. W odróżnieniu od filtra `route` są to więc także informacje z protokołów wyższych niż TCP i UDP – numery portów, typy pakietów ICMP itp.

4.3 u32

Najbardziej złożony z filtrów dostępnych w Linuxie. Jest w całości oparty o tablice haszujące, co zapewnia wydajność nawet przy bardzo złożonych zbiorach reguł. Posiada największe możliwości jeśli chodzi o wybór kryteriów, które muszą spełniać klasyfikowane pakiety. Filtr `u32` jest opisany w rozdziale 10, strona 20.

5 Elementarne algorytmy kolejkowania

Kernel Linuxa obsługuje następujące elementarne algorytmy kolejkowania:

5.1 FIFO

Kolejka **FIFO** (*First In, First Out*, nazywana także *drop-tail*). Najczęściej stosowana, nie tylko zresztą w ruterach. Prosta kolejka pakietów, przesuujących się do wyjścia. Limitowana wyłącznie przez wydajność interfejsu wyjściowego. Jedynym jej parametrem jest wielkość, mierzona w bajtach dla `bfifo` (*byte FIFO*) lub pakietach dla `pfifo` (*packet FIFO*).

5.2 TBF

Algorytm **TBF** (*Token Bucket Filter*) to prosta kolejka wypuszczająca wyłącznie pakiety poniżej ustalonego administracyjnego natężenia przepływu, z możliwością buforowania chwilowych przeciążeń.

Implementacja TBF posiada bufor (kubek, *bucket*), do którego wpadają żetony z natężeniem (*token rate*) określonym parametrem `rate`. Rozmiar kubka (ilość żetonów, które może pomieścić) jest określany parametrem `buffer`.

Każdy żeton wypuszcza z kolejki określoną ilość danych i równocześnie z jej wysłaniem jest on usuwany z kubka. W ten sposób możliwe są następujące sytuacje:

- Dane wchodzi do TBF z natężeniem przepływu **równym** natężeniu przepływu żetonów. Wówczas każda porcja danych ma swój odpowiadający żeton i przechodzi przez filtr.
- Dane wchodzi do TBF z natężeniem przepływu **mniejszym** niż żetony. Ponieważ tylko część żetonów jest „zabierana” przez wychodzące dane, kubek wypełnia się niewykorzystanymi żetonami (ale nie może ich być więcej niż `buffer`).

Mogą zostać one wykorzystane w przyszłości, w przypadku chwilowego przeciążenia (*burst*).

- Jeśli natężenie danych jest **większe** niż ustalone natężenie żetonów, to mamy do czynienia z przeciążeniem filtra. W tej sytuacji dane mogą być wysyłane dopóki nie zostaną zużyte wszystkie żetony, które mogły się tam nagromadzić w okresie poprzedzającym przeciążenie. Jeśli w kubelku nie ma już żetonów, pakiety są kasowane.

Jak wynika z powyższego opisu, parametr **buffer** określa, jak długo TBF będzie w stanie buforować dane przekraczające parametr **rate** w wypadku przeciążenia. Drugi istotny parametr to **limit**, określający wielkość kolejki w bajtach.

Algorytm TBF znajduje powszechne zastosowanie do ograniczania pasma oraz regulowania natężenia przepływu danych w protokołach typu RSVP i innych, służących do gwarantowania przydziału zasobów sieciowych (patrz RFC 2215 [4]).

5.3 PRIO

PRIO (*Simple Priority Queueing*) to prosta kolejka umożliwiająca preferowanie określonych pakietów. Składa się z kilku kolejek, z których zawsze najpierw obsługiwane są te o wyższym priorytecie. W razie przekroczenia limitu wielkości kolejek, pierwsze kasowane są pakiety z kolejki o najniższym priorytecie. Każda niższa kolejka jest obsługiwana dopiero wówczas, gdy pakiety z wyższych kolejek zostaną wysłane.

Implementacja linuksowa została zaprojektowana w dość interesujący sposób, a mianowicie priorytety pakietów są obliczane na podstawie pola TOS (*Type of Service*) każdego przetwarzanego pakietu.

Oznacza to, że kolejka **PRIO** automatycznie będzie preferować oznaczone w TOS jako interaktywne (*telnet*, *SSH*) nad danymi masowymi. To czyni ją bardzo przydatną szczególnie na wolnych łączach lub słabych sprzętowo routerach, gdzie zastosowanie bardziej zaawansowanych technik albo nie ma sensu, albo nie jest możliwe z powodu wymagań odnośnie precyzji zegara.

5.4 SFQ

SFQ (*Stochastic Fairness Queueing*) to prosta i posiadająca niewielkie wymagania obliczeniowe odmiana algorytmu „sprawiedliwego kolejkowania” (*fair queueing*). Kolejka jest w tym algorytmie rozpatrywana jako składająca się z ciągów pakietów zwanych konwersacjami (*conversations*) lub strumieniami (*flows*). Do jednej konwersacji należą pakiety posiadające takie same adresy źródłowe i docelowe oraz protokół w nagłówku IP. Przykładowo, do samodzielnych konwersacji należeć będą połączenie TCP, ciągły strumień danych UDP (NFS, Quake) między dwoma hostami itp.

Każda z konwersacji jest obsługiwana sprawiedliwie i cyklicznie (*round-robin*), to znaczy w każdym przebiegu wysyłane jest po jednym pakiecie z każdego ciągu.

Konsekwencją jest również to, że najszybciej obsługiwane są konwersacje „krótkie”, czyli stanowiące małe obciążenie dla sieci. W przypadku przepełniania natomiast gubione są pakiety z końca kolejki (stąd *tail-drop*).

SFQ nie potrafi jednak odróżnić danych interaktywnych od masowych. Jeśli zachodzi taka potrzeba, selekcję należy przeprowadzić wcześniej, używając SFQ jako algorytmu kolejkowania dla podklas CBQ przeznaczonych dla ruchu masowego. SFQ zapewnia tylko sprawdzieliwe współdzielenie łącza przez kilka aplikacji, zwiększa przewidywalność czasów wędrówki pakietów konwersacji o dużym natężeniu i zapobiega przejściu całego pasma przez jedną aplikację generującą duży ruch (*link take-over*).

Parametrami SFQ są: częstość przeliczania funkcji haszującej (**perturb**) oraz jednostka kolejkowania (**quantum**).

Ta ostatnia wartość powinna być większa lub równa MTU obowiązującemu na danym interfejsie. Regularne przeliczanie funkcji haszującej jest konieczne ze względu na jej prostotę, która może powodować kolizje czyli identyczne wyniki dla różnych konwersacji. Parametr ten podaje się w sekundach i powinien być uzależniony od natężenia oraz charakterystyki ruchu – im więcej podobnych konwersacji przechodzi przez łącze, tym częściej powinna być przeliczana funkcja haszująca. W praktyce stosuje się czasy w przedziale 5–20 sekund.

5.5 RED

RED (*Random Early Detection*) to algorytm mający na celu unikanie przeciążeń przez wykorzystanie mechanizmów istniejących już w protokole TCP. RED „przewiduje” wystąpienie przeciążenia łącza i gubi pakiety, sygnalizując tym samym nadawcy, że powinien ograniczyć transmisję. W przypadku protokołu TCP jest to założenie jak najbardziej poprawne, a po zakończeniu przeciążenia TCP potrafi automatycznie powrócić do optymalnej prędkości wysyłania pakietów. W związku z tym RED znajduje zastosowanie jako algorytm w podklasach CBQ, przeznaczonych dla ruchu TCP.

Charakterystyczną cechą RED jest to, że – w odróżnieniu od pozostałych algorytmów – próbuje on zapobiegać przeciążeniu zanim ono wystąpi, a nie tylko zminimalizować jego skutki w czasie jego trwania. Pierwszym krokiem po przyjęciu nowego pakietu przez RED jest obliczenie średniego rozmiaru kolejki (w bajtach) będącego funkcją poprzedniej średniej oraz obecnego rozmiaru. Na tej podstawie – oraz dwóch administracyjnie ustalonych parametrów **min** i **max** – RED oblicza prawdopodobieństwo z jakim pakiet powinien zostać odrzucony. Prawdopodobieństwo to rośnie wraz ze wzrostem nasycenia łącza, aż do porzucenia pakietu.

Algorytm RED jest opisany m. in. w RFC 2309 [1].

5.6 CBQ

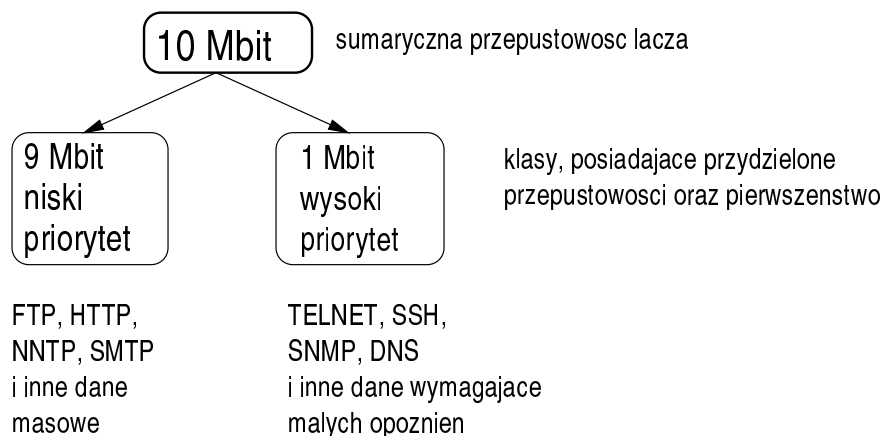
CBQ (*Class Based Queueing*) jest algorytmem, stanowiącym podstawę do podziału przepustowości łącza (*link sharing*) oraz szkielet pozwalający na wykorzystanie wszystkich wyżej opisanych algorytmów elementarnych.

W CBQ ruch jest dzielony na kilka kolejek, zwanych dalej klasami, charakteryzujących się priorytetem oraz przydzieloną przepustowością. Siłą CBQ jest to, że pakiety mogą być rozdzielane do kolejek-klas na podstawie dowolnych kryteriów za pomocą opisanych wcześniej filtrów. Natomiast zamiast domyślnie ustawianych kolejek FIFO, składowymi CBQ mogą być dowolne z również opisanych wyżej algorytmów elementarnych.

Cechy te pozwalają na osiągnięcie następujących efektów:

- Podział sumarycznej przepustowości łącza na kilka części (klas), przydzielonych według potrzeb określonym rodzajom usług, adresom IP itp.
- Przesyłanie pakietów z różnym pierwszeństwem w zależności od tych samych parametrów.
- Przydzielenie odpowiednim rodzajom ruchu właściwych algorytmów kolejkowania, np. dla ruchu masowego – algorytmu SFQ, dla ruchu TCP – algorytmu RED itp.

Przykładowa struktura klas może wyglądać następująco:



W powyższym przykładzie, przepustowość 10 Mbit została podzielona na dwie części (klasy CBQ). Pierwszej z nich przydzieliliśmy 1 Mbit oraz wysoki priorytet (małe opóźnienie w razie przeciążenia). Przez tę klasę przesyłane są protokoły interaktywne typu TELNET lub SSH, które transportują niewielkie ilości danych, ale wymagają bardzo małych opóźnień na łączu. Druga klasa – przeznaczona dla danych

masowych – otrzymała 9 Mbit pasma, ale niższy priorytet. W ten sposób, nawet jeśli danym łączem jeden użytkownik będzie ściągał przez FTP duży plik, to drugi, wykorzystujący w tym czasie przez TELNET, powinien móc dalej pracować. Bez podziału pasma płynące z dużym natężeniem dane protokołu FTP prawdopodobnie nasyciłyby łącze w 100% i praktycznie uniemożliwiły pracę użytkownikowi TELNETa.

Warto zaznaczyć, że powyżej ruch do klas jest rozdzielany według protokołów. Możliwe jest rozdzielanie (klasyfikacja) pakietów według praktycznie dowolnych kryteriów – takich jak adresy IP (przydział pasma dla określonych hostów), pole TOS (pozwala wykorzystać istniejący mechanizm określania pierwszeństwa pakietów) i inne.

Dodatkowo, CBQ umożliwia optymalne wykorzystanie łącza przez „pożyczanie” pasma przez aktualnie obciążoną klasę od innej, nieobciążonej. Zachowanie to można kontrolować, zabraniając określonej klasie pożyczania pasma (parametr **bounded**) lub uniemożliwiając pożyczanie z określonej klasy (parametr **isolated**).

Jak już napisaliśmy, klasy są kanałami logicznymi do których rozdzielany jest ruch wchodzący do złożonej kolejki. Każda klasa posiada przyporządkowany elementarny algorytm kolejkowania, który odpowiada za wypuszczanie danych znajdujących się w danej klasie.

Klasy CBQ charakteryzuje kilka podstawowych parametrów, które są poniżej opisane w składni stosowanej przez polecenie **tc**:

- **parent** – identyfikator kolejki macierzystej, do której dana klasa przynależy. Podawany w postaci **KOLEJKA:0**.
- **classid** – identyfikator danej klasy podawany w postaci **KOLEJKA:KLASA**, gdzie **KOLEJKA** jest numerem kolejki do której dana klasa przynależy, a **KLASA** – numerem deklarowanej klasy. Ponieważ kolejka macierzysta jest i tak podawana w parametrze **parent**, wystarczy podać sam numer klasy, np. **classid :3**.
- **prio** – priorytet danej klasy, podawany jako liczba całkowita z przedziału 1...10. Im mniejsza wartość, tym większe pierwszeństwo będzie miała dana klasa. Priorytet pozwala podzielić klasy według wzrastającego pierwszeństwa w obsłudze danych przekazywanych przez daną klasę. Przykładowo, klasom przenoszącym ruch interaktywny (TELNET, SSH) można ustawić większy priorytet niż klasom z ruchem masowym (FTP). Oczywiście, liczbowe wartości priorytetów dla klas interaktywnych powinny być mniejsze niż dla klas masowych.
- **bandwidth** – sumaryczna przepustowość interfejsu, w którym konfigurowane są klasy. Podawana jako liczba z przyrostkiem określającym jednostkę – np. dla Ethernetu 10Base-T **10mbit**. Inne dopuszczalne jednostki to **bps**, **kbits** (odpowiednio bajty oraz kilobajty na sekundę, **kbit** oraz **mbit** (kilobity i megabity na sekundę).
- **rate** – przepustowość pasma przydzielonego danej klasie, stanowiąca ułamek przepustowości interfejsu. Jednostki są identyczne jak w parametrze **bandwidth**.

Przykładowo, jeśli wybranej klasie skonfigurowanej na interfejsie typu Ethernet chcemy przydzielić 40% pasma, parametr ten będzie miał postać `rate 4mbit`.

- **weight** – względna waga danej klasy, która dla wszystkich klas powinna być wartością proporcjonalną do **rate**. W praktyce można przyjmować wartości dziesięciokrotnie mniejsze niż **rate**, ale parametr ten nie jest niezbędny i można zawsze używać wartości 1.
- **avpkt** – średnia wielkość pakietów przesyłanych w tej klasie, w bajtach. Wielkość tę można wyznaczyć eksperymentalnie, ale przeważnie będzie ona stanowić 50-60% MTU danego interfejsu.
- **mput** – minimalna wielkość pakietów, które będą przesyłane przez daną klasę, w bajtach. Mniejsze pakiety nie będą podlegały klasyfikacji.
- **allot** – suma MTU oraz nagłówka warstwy łącza danego interfejsu w bajtach. Przykładowo, dla zwykłego Ethernetu (10Base-T) wartość ta wynosi 1514 (1500 + 14 nagłówka Ethernet), a dla połączenia PPP z MTU 576 – 580 (576 + 4 bajty nagłówka PPP).
- **maxburst** – parametr określający dopuszczalne chwilowe przeciążenie danej klasy (*burstiness*). Parametr ten jest ściśle związany z samym algorytmem CBQ i szczegółowo opisany w literaturze na ten temat [2], [3].
- **est** – dwa parametry miernika natężenia ruchu (*rate estimator*) pracującego w danej klasie, podawane w postaci **est X Y**. Miernik określa średnie natężenie przepływu danych w danej klasie przez okres X sekund ze stałą czasową Y sekund. W praktyce typowymi wartościami jest 1 i 8 – **est 1sec 8sec**.

6 Rozdzielczość zegara

Wszystkie wymienione powyżej algorytmy w znaczym stopniu polegają na dokładnym odmierzaniu czasu przez jądro, od czego bezpośrednio zależą takie parametry jak maksymalne i minimalne natężenia przepływu możliwe do obsłużenia przez kolejki, dokładność przycinania pasma itp. Linux może wykorzystywać trzy źródła czasu, konfigurowalne za pomocą deklaracji na początku pliku `/usr/src/linux/include/net/pkt_sched.h`:

- **PSCHED_GETTIMEOFDAY** – najprostsza metoda, opierająca się o wywołanie systemowe `gettimeofday`, przy tym najmniej doskonała. Nie zapewnia zbyt dużej dokładności oraz rozdzielczości zegara.

- **PSCHED_JIFFIES** – źródłem taktu zegara są *jiffies*, czyli podstawowe jednostki odmierzenia czasu używane przez jądro. W praktyce wystarczające do większości zastosowań, nie pozwalają na osiągnięcie zbyt dużej dokładności w przycinaniu pasma.
- **PSCHED_CPU** – źródłem taktu jest sam procesor, co daje największą dokładność i rozdzielczość zegara, ale wymaga wydajnego procesora – Alpha lub Pentium posiadające instrukcję **RTDSC**. W chwili obecnej ma ją większość stosowanych procesorów, co można dodatkowo sprawdzić w `/proc/cpuinfo`, powinna być ustawiona flaga `tsc`).

7 Proste przykłady zastosowania kolejek

7.1 Domyślne ustawienia

Przed eksperymentami z QoS warto zobaczyć jakie są domyślne ustawienia interfejsów tworzonych przez jądro. Wykorzystamy do tego celu polecenie `ip link show dev DEV`, gdzie `DEV` jest nazwą interfejsu. Poniżej wyniki z kilku wybranych interfejsów naszego routera:

```
2: eth0: <BROADCAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
30229: ppp1: <POINTOPOINT,NOARP,UP> mtu 1500 qdisc pfifo_fast qlen 10
30969: ppp11: <POINTOPOINT,NOARP,UP> mtu 576 qdisc pfifo_fast qlen 10
```

Jak widać, wszystkie interfejsy używają algorytmu *packet FIFO*, różniąc się przy tym wielkością MTU (wartość ustawiana administracyjnie, domyślnie jest to 1500 bajtów dla Ethernetu oraz połączeń PPP) oraz długością kolejki. Ten ostatni parametr wynosi 100 pakietów dla szybkiego Ethernetu, oraz 10 dla relatywnie dużo wolniejszych łączy PPP.

7.2 Prosta kolejka PRIO

Najprostsza, prawie całkowicie automatycznie konfigurowana kolejka PRIO. Poniższy przykład stworzy PRIO składające się z trzech kolejek:

```
$ tc qdisc add dev ppp0 root prio
$ tc -s qdisc
qdisc prio 8017: dev ppp0 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
Sent 7403354 bytes 82060 pkts (dropped 96, overlimits 0)
```

Parametr `priomap` oznacza jakie wartości pola TOS są przypisane do której z trzech kolejek oznaczonych 0 – 2. Przyporządkowanie to odbywa się na podstawie 16-to pozycyjnej tablicy, mapującej wartości TOS na numery kolejek. Tablica ta jest zaimplementowana na stałe (można ją znaleźć w źródłach kernela), ale można układać własne mapowanie korzystając właśnie z parametru `priomap`.

7.3 Ograniczenie pasma (TBF)

Najprostszy przykład zastosowania filtra *token bucket* do ograniczenia przepustowości interfejsu do określonej wartości. W opisywanym przypadku ograniczenie do 128 kbit nałożymy na interfejs `eth0`.

```
$ tc qdisc add dev eth0 root handle 1: tbf buffer 10KB limit 10KB \
    rate 128kbit
```

Parametry TBF zostały omówione już wcześniej. Parametr `root` oznacza, że dodawana kolejka jest główną dla tego interfejsu, z identyfikatorem `1:`. Wykorzystajmy teraz polecenie `tc` do wyświetlenia parametrów kolejki obecnie obowiązującej na tym interfejsie:

```
$ tc -s qdisc
qdisc tbf 1: dev eth0 rate 128Kbit burst 10Kb lat 0us
Sent 26386 bytes 180 pkts (dropped 0, overlimits 0)
```

Teraz, intensywnie obciążając interfejs ruchem wychodzącym (sic!) możemy obserwować TBF w akcji:

```
$ tc -s qdisc
qdisc tbf 1: dev eth0 rate 128Kbit burst 10Kb lat 0us
Sent 621683 bytes 621 pkts (dropped 19, overlimits 2014)
backlog 7722b 7p
```

Powyższe dane zostały wyświetlone w momencie, gdy łącze było nadal obciążone. Widać, że 2014 pakietów przekroczyło narzucony limit 128 Kbit, ale do tej pory skasowanych zostało tylko 19 z nich. Reszta z nich została zbuforowana przez czas trwania przeciążenia (w tym wypadku krótkotrwałego). Pakiety aktualnie przechowywane w buforze pokazuje parametr `backlog`. Do obciążania łącza używałem programu `ttcp` [5], który działa w trybie klient-serwer i wyświetla zmierzoną średnią przepustowość łącza. W tym wypadku `ttcp` pokazało wynik 113.42 Kbit, co mniej więcej odpowiada ustawionemu limitowi. Na koniec pozostaje już tylko posprzątać po sobie poleceniem `tc qdisc del root dev eth0`. Spowoduje to usunięcie modułu TBF z interfejsu i przywrócenie domyślnego FIFO.

7.4 Automatyczne współdzielenie łącza (SFQ)

Poprzedni przykład narzucał na interfejs sztuczny limit przepustowości. W tym wypadku cel będzie dokładnie przeciwny – poprawienie pracy łącza przez wykorzystanie algorytmu SFQ, opisanego szczegółowo wcześniej. Tak jak poprzednio, SFQ będzie głównym algorytmem kolejującym danego interfejsu

```
$ tc qdisc add dev eth0 root handle 1: sfq perturb 5 quantum 1500b
$ tc -s qdisc
qdisc sfq 1: dev eth0 quantum 1500b perturb 5sec
  Sent 11498 bytes 74 pkts (dropped 0, overlimits 0)
```

Tym razem uzyskanie zauważalnych wyników byłoby dużo trudniejsze, bo działanie SFQ polega na wyrównywaniu wykorzystania łącza przez kilka równoległych połączeń, a nie na przykład preferowanie jednego z nich.

SFQ bardzo dobrze nadaje się na algorytm ogólnego przeznaczenia, poprawiający współdzielenie łącza przez wielu użytkowników lub aplikacji i zapobiegający przejęciu pasma przez jedną transmisję. Dodatkową zaletą jest prosta konfiguracja, nie wymagająca dokładnego zaplanowania podziału pasma między poszczególnych użytkowników.

8 Podział łącza na klasy za pomocą CBQ

W wielu przypadkach zachodzi potrzeba dokładniejszego określenia, w jaki sposób ma być podzielona sumaryczna przepustowość łącza, jakie usługi potrzebują zarezerwowanej części pasma, które z nim mają być obsługiwane z wyższym, a które z niższym pierwszeństwem. W tym wypadku automatyczne segregowanie ruchu zapewniane przez SFQ nie wystarcza i konieczne jest zbudowanie struktury klas CBQ.

Tym razem przykład dotyczy łącza asynchronicznego PPP, o przepustowości 115.2 kbit/sek i MTU ustawionym na 1500 bajtów. Parametry poniższych poleceń zostały dobrane eksperymentalnie, na podstawie eksperymentów na działającym łączu.

W pierwszym kroku tworzymy złożoną kolejkę o identyfikatorze 1:0, na interfejsie ppp0. Średnia wielkość pakietu `avpkt` to 1000 bajtów. Na tym poziomie wielkość ta nie ma aż takiego znaczenia, ważne jest by była mniejsza od MTU łącza.

```
$ tc qdisc add dev ppp0 root handle 1:0 cbq bandwidth 115200 avpkt \
  1000 mpu 64
```

Tworzymy klasę 1:1 dla ruchu interaktywnego i wymagającego małych opóźnień. Klasie tej przydzielamy 28700 bit/sek przepustowości oraz wysoki priorytet 2. Ustawiamy jej także trochę większą tolerancję na chwilowe przeciążenia (`maxburst`) oraz mniejszy średni pakiet (`avpkt`). Do klasy tej również podpinamy algorytm SFQ.

```
$ tc class add dev ppp0 parent 1:0 classid 1:1 est 2sec 16sec cbq \
  bandwidth 115200 rate 28700 allot 1504b weight 1 prio 2 \
  maxburst 10 avpkt 512
$ tc qdisc add dev ppp0 parent 1:1 sfq quantum 1500b perturb 5
```

Tworzymy klasę 1:2 przeznaczoną dla ruchu masowego TCP, przydzielając jej 69110 bit/sek z sumarycznej przepustowości łącza oraz priorytet 6 – czyli dość niski.

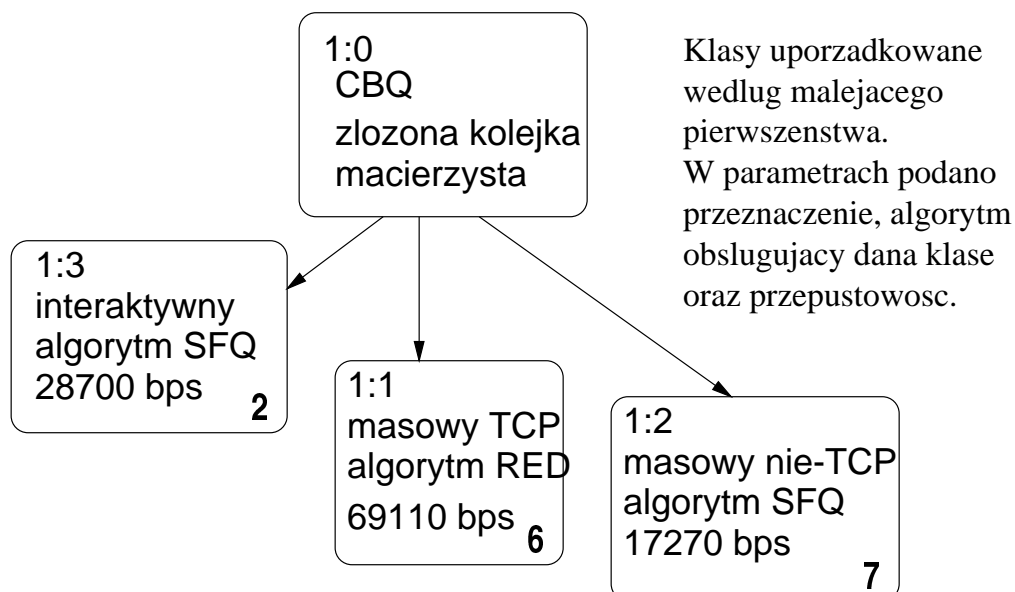
Parametr `avpkt` jest ustawiony na 1000 bajtów, ponieważ w ruchu masowym dominować będą prawdopodobnie większe pakiety. Do klasy tej podpinamy algorytm RED, zaprojektowany do protokołu TCP.

```
$ tc class add dev ppp0 parent 1:0 classid 1:2 est 1sec 8sec cbq \
  bandwidth 115200 rate 69110 allot 1504b weight 1 prio 6 \
  maxburst 5 avpkt 1000
$ tc qdisc add dev ppp0 parent 1:2 red limit 20KB min 5KB max 15KB \
  burst 20 avpkt 1000 bandwidth 115200 probability 0.4
```

Tworzymy klasę 1:3, do której trafiać mają pakiety z protokołów innych niż TCP, ale nie przenoszące usług interaktywnych. Przydzielamy jej pasmo 17270 bit/sek oraz priorytet 7 – jeszcze niższy niż dla klasy 1:1. Istotny jest parametr `defmap`, stanowiący maskę dla priorytetów logicznych pakietów, ustawianych przez jądro. W tym wypadku oznacza on, że dana klasa jest klasą domyślną i do niej trafiają pakiety nie skierowane przez filtry do innych klas. Do klasy 1:3 podpinamy algorytm SFQ.

```
$ tc class add dev ppp0 parent 1:0 classid 1:3 est 1sec 8sec cbq \
  bandwidth 115200 rate 17270 allot 1504b weight 1 prio 7 \
  maxburst 5 avpkt 1000 defmap 3f
$ tc qdisc add dev ppp0 parent 1:3 sfq quantum 1500b perturb 5
```

Podsumowując, mamy teraz następujące klasy:



Od tej pory ruch wychodzący przez dany interfejs jest obsługiwany przez algorytm CBQ, ale wykorzystywana jest wyłączenia jedna klasa, oznaczona jako domyślna – czyli 1:3. Żeby wykorzystać pozostałe klasy, musimy skierować do nich pakiety wyselekcjonowane za pomocą filtrów. Poniżej przedstawimy trzy – funkcjonalnie identyczne – konfiguracje wykonane za pomocą różnych filtrów. Konfiguracja jest następująca:

1:1 Klasa przeznaczona dla ruchu interaktywnego.

Przydzielamy do niej protokoły TELNET, SSH oraz wszystkie posiadające pole TOS ustawione na 0x10 (małe opóźnienie). Dodatkowo do tej klasy przydzielamy ruch wysyłany do sieci 192.168.1.0/24 – dla większego realizmu można przyjąć, że jest to podsieć, w której znajdują się komputery dyrekcji, a tak na prawdę chodzi tylko o okazję do pokazania możliwości filtra `route`.

1:2 Klasa przeznaczona dla ruchu masowego TCP.

Do tej klasy przydzielamy cały ruch TCP/IP, który nie został zakwalifikowany do klasy interaktywnej.

1:3 Klasa przeznaczona dla ruchu masowego innego niż TCP.

Do tej klasy nic specjalnie nie przydzielamy, bo automatycznie trafi do niej wszystko co nie zostanie zaklasyfikowane do klas 1:1 i 1:3.

8.1 Klasyfikacja filtrem fw

Do konfiguracji filtra `fw` służy poleceni `ipchains`. Tworzymy standardowe regułki filtra pakietów `ipchains`, z tą różnicą że do każdej z nich dodajemy parametr `-m 0x1000x` określający klasę do której ma być skierowany pasujący pakiet. Cyferka `x` oznacza numer klasy – 1:x. Zgodnie z powyższymi założeniami, powinniśmy stworzyć następujące regułki:

```
# 1:1
$ ipchains -A output -p tcp -b -s 0.0.0.0/0 telnet -m 0x10001 -j ACCEPT
$ ipchains -A output -p tcp -b -s 0.0.0.0/0 22 -m 0x10001 -j ACCEPT
$ ipchains -A output -p ip -d 192.168.1.0/24 -m 0x10001 -j ACCEPT
# 1:2
$ ipchains -A output -p tcp -m 0x10002 -j ACCEPT
$ tc filter add dev ppp0 parent 1:0 protocol ip prio 90 fw
```

Filtr `ipchains` nie pozwala na dopasowywanie pakietów po polu TOS, więc ten parametr w tym wypadku pomijamy. Opcji `-b` używamy dla uproszczenia konfiguracji – zamiast pisać regułkę dwukrotnie, dla pakietów pochodzących z portu TELNET i do niego skierowanych.

8.2 Klasyfikacja filtrem route

W przypadku filtra `route` nie ma w ogóle mowy o rozdzielaniu ruchu według protokołów warstwy transportowej (TCP, UDP). Filtr ten umożliwia klasyfikację pakietów według parametrów nagłówka IP, takich jak adres źródłowy, adres docelowy oraz pole TOS. Tym razem pominiemy więc klasyfikację protokołów TELNET i SSH, klasyfikując pakiety przeznaczone dla podsieci 192.168.1.0/24 oraz według pola TOS:

```
$ ip rule add to 192.168.1/24 realms dyrekcja
$ ip rule add tos 0x10 realms dyrekcja
$ tc filter add dev ppp0 parent 1:0 protocol ip prio 100 route \
  to dyrekcja flowid 1:1
```

Konfiguracja filtra `route` zmieniła się od wczesnych wersji jąder 2.2 o tyle, że przyporządkowanie numeru klasy nie odbywa się od razu na poziomie regułek `ip rule` (jak było kiedyś). Zamiast tego w regułkach tych określamy królestwo (*realm*), do którego przynależy określony ruch. Przyporządkowanie klasy do królestwa odbywa się na etapie konfiguracji samego filtra (ostatnia linijka).

Warto zaznaczyć, że możliwe jest połączenie filtra `fw` z filtrem `route` w taki sposób, że najpierw filtr pakietów znakuje pakiety a następnie filtr `route` przyporządkowuje pakiety z określonym oznaczeniem do poszczególnych królestw (parametr `fwmark` polecenia `ip rule`). Z drugiej strony, bezpośrednie wykorzystanie filtra `fw` do tego celu jest prostsze.

8.3 Klasyfikacja filtrem u32

Filtr `u32` łączy w sobie funkcje filtrów `route` i `fw`, umożliwiając klasyfikację pakietów według większości informacji, jakie tylko da się wyciągnąć z nagłówka pakietu IP, TCP oraz UDP. Konfiguracja tego filtra dla naszej przykładowej kolejki CBQ jest pozbawiona ograniczeń, które posiadały poprzednie filtry.

Na początku inicjalizujemy filtr z szesnastoma pozycjami w tablicy haszującej:

```
$ tc filter add dev ppp0 parent 1:0 prio 10 protocol ip u32 divisor 16
```

Dodajemy regułkę dla pakietów IP posiadających pole TOS ustawione na 0x10 (minializacja opóźnień):

```
$ tc filter add dev ppp0 parent 1:0 prio 10 u32 \
  match ip tos 0x10 0xff \
  flowid 1:1
```

Dodajemy regułki klasyfikujące pakiety TCP należące do protokołów TELNET (port źródłowy lub docelowy 0x17) oraz SSH (port 0x16):

```
$ tc filter add dev ppp0 parent 1:0 prio 10 u32 \
    match tcp dst 0x17 0xffff \
    flowid 1:1
$ tc filter add dev ppp0 parent 1:0 prio 10 u32 \
    match tcp src 0x17 0xffff \
    flowid 1:1
$ tc filter add dev ppp0 parent 1:0 prio 10 u32 \
    match tcp dst 0x16 0xffff \
    flowid 1:1
$ tc filter add dev ppp0 parent 1:0 prio 10 u32 \
    match tcp src 0x16 0xffff \
    flowid 1:1
```

Dodajemy regułę klasyfikującą pakiety IP skierowane do sieci 192.168.1.0/24:

```
$ tc filter add dev ppp0 parent 1:0 prio 10 u32 \
    match ip dst 192.168.1/24 \
    flowid 1:1
```

Na końcu dodajemy regułę klasyfikującą ruch IP, który nie został dopasowany do poprzednich reguł:

```
$ tc filter add dev ppp0 parent 1:0 prio 10 u32 \
    match ip protocol 0x6 0xff \
    flowid 1:2
```

Konfiguracja jest, jak widać, bardziej skomplikowana niż dla poprzednio omawianych filtrów, ale wynika to z przyjętej na etapie projektowania elastyczności filtra u32.

9 Dodatki

9.1 Jednostki przepustowości łącz

Wewnętrzną jednostką przepustowości łącza oraz natężenia ruchu, stosowaną przez algorytmy kolejowania w Linuxie jest **bps** – bajt na sekundę. Program **tc** umożliwia jednak podawanie wartości w innych, powszechnie stosowanych jednostkach, wymienionych poniżej. Wszystkie one są przeliczane na **bps** zgodnie z podanym współczynnikiem:

Jednostka	Pełna nazwa	Przelicznik do bps
(brak)	bity na sekundę	$\frac{1}{8}$
bps	bajty na sekundę	1
kbps	kilobajty na sekundę	1024
mbps	megabajty na sekundę	$1024 \cdot 1024$
kbit	kilobity na sekundę	$1024 \cdot \frac{1}{8}$
mbit	megabity na sekundę	$1024 \cdot 1024 \cdot \frac{1}{8}$

Do wyświetlania tych wartości, program `tc` wykorzystuje natomiast wyłączenie jednostki `mbit`, `kbit` oraz `bps` wybierane dla różnych przedziałów przepustowości. Przykładowo, w poniższym przykładzie prędkość ustawiamy w bitach na sekundę (57600 bit/sek), ale wyświetlona zostanie w bajtach na sekundę (7200 bajtów/sek):

```
$ tc qdisc add dev eth0 root handle 1: tbf limit 10K burst 10K rate
57600
$ tc -s qdisc
qdisc tbf 1: rate 7200bps burst 10Kb lat 1us
Sent 145671 bytes 423 pkts (dropped 21, overlimits 2392)
backlog 4732b 14p
```

10 Filtr u32

Filtr `u32` w najprostszej wersji jest listą, zawierającą rekordy składające się z dwóch pól: selektora i akcji. Selektory, opisane poniżej są dopasowywane kolejno do aktualnie przetwarzanego pakietu IP. Pasujący selektor pociąga za sobą wykonanie przypisanej do niego akcji. Najprostszym przypadkiem akcji jest skierowanie bieżącego pakietu do określonej klasy CBQ.

Polecenie `tc filter` służące do konfiguracji filtra `u32` składa się z trzech części: specyfikacji filtra, selektora oraz akcji. We wszystkich zamieszczonych w tym dokumencie przykładach użycia filtra `u32` części te znajdowały się w osobnych liniach. Podstawowa specyfikacja filtra wygląda następująco:

```
tc filter add dev INTERFEJS [ protocol PROTO ]
                        [ (preference|priority) PRIO ]
                        [ parent CBQ ]
```

Pole `protocol` określa protokół, który będzie przetwarzany przez filtr. Teoretycznie może to być każdy protokół warstwy sieci, obsługiwany przez Linuxa, w praktyce stosuje się jedynie `protocol ip`. Opcja `preference` określa numer kolejny – a tym samym pierwszeństwo – filtra, jeśli zdefiniowano ich kilka. Opcja `parent` określa, do jakiej kolejki CBQ odnosi się dany filtr. Jej parametrem jest identyfikator danej kolejki.

Powyżej opisane parametry odnoszą się do wszystkich rodzajów filtrów, nie tylko `u32`.

10.1 Selektor u32

Selektor u32 zawiera informację o tym, jakie pola w nagłówkach pakietów, o określonej długości i przesunięciu, mają pasować do podanego wzorca. Najprościej unaocznic to na przykładzie. Spójrzmy na wynik polecenia `tc filter ls dev ppp0` wydany dla już skonfigurowanego, dość skomplikowanego filtra:

```
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
```

Pierwsza z nich pasuje do pakietów IP, które w drugim bajcie mają liczbę 0x10, przy czym porównywane jest pole o długości jednego bajta (0xff). Drugi bajt pakietu IP to pole TOS. Jest to więc dokładny odpowiednik naszej reguлки `match ip tos 0x10 0xff` przetworzony na wewnętrzny format filtra u32. Parametr `at` określa przesunięcie względem początku pakietu, więc w tym wypadku wypadło na drugi bajt.

Przyjrzyjmy się kolejnej regułce:

```
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
```

Opcja `nexthdr` oznacza kolejny nagłówek, enkapsulowany w przetwarzanym pakiecie IP. Tym razem również zaczynamy liczenie od początku pakietu (+0). Dopasowanie ma nastąpić w drugim, 32-bitowym słowie nagłówka. W protokole TCP i UDP jest to pole zawierające numer portu przeznaczenia pakietu. Ponieważ w obowiązującym w Internecie formacie jest kolejność `big--endian` (starsze bity na początku), liczbę tę należy czytać jako 0x0016, czyli 22 decymalnie. Regułka ta będzie zatem pasować do pakietów protokołu SSH (przesyłanych po SSH) oraz pakietów UDP wysyłanych do portu 22 (ten port UDP nie jest na razie obsadzony przez żadną popularną usługę). Dokładne wyjaśnienie tego problemu znajduje się poniżej, w opisie parametrów konfiguracyjnych filtra u32.

Ostatnia regułka odnosi się znowu do samego nagłówka IP:

```
filter parent 1: protocol ip pref 10 u32 fh 800::807 order 2055 key ht
800 bkt 0 flowid 1:4
    match c0a80100/ffffff00 at 16
```

Widzimy tutaj dopasowanie trzech bajtów, począwszy od 16-tego bajtu nagłówka. Jest to pole zawierające adres przeznaczenia pakietu IP. Jest to nasza poprzednia regułka klasyfikująca pakiety IP po adresie – `match ip dst 192.168.1/24`.

10.2 Parametry ogólne

Parametry ogólne definiują wzorzec, maskę oraz pozycję, na której mają być one dopasowywane do zawartości datagramu IP. Składnia selektora `u32` jest dla parametrów ogólnych następująca:

```
match [ u32 | u16 | u8 ] WZÓR MASKA [ at OFF | nexthdr+OFF]
```

Jeden z selektorów `u32`, `u16` i `u8` określa długość wzorca (odpowiedni 32, 16 i 8 bitów). Po tym następują `WZÓR` oraz `MASKA` o długości określonej przez selektor. Parametr `at OFF` oznacza przesunięcie w bajtach względem początku pakietu, a `nexthdr+OFF` – analogiczne przesunięcie względem początku pakietu wyższej warstwy enkapsulowanego w przetwarzanym datagramie.

Przykłady:

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

Pakiet pasuje do powyższej regułka, jeśli jego czas życia (TTL) wynosi 64. TTL jest polem zaczynającym się po 8 bajcie nagłówka IP.

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

Ta z kolei regułka będzie pasowała tylko do pakietów TCP z ustawionym bitem ACK. Komentarza wymaga pierwszy z użytych selektorów – bit ACK jest drugim najstarszym bitem (`0x10`) w 14-tym bajcie nagłówka TCP (`at nexthdr+13`). Gdybyśmy lubili utrudniać sobie życie, zamiast prostego `protocol tcp` moglibyśmy na przykład napisać `match u8 0x06 0xff at 9`, bo 6 jest numerem protokołu TCP, a numer ten jest umieszczony w nagłówku IP w 10-tym bajcie.

10.3 Parametry szczegółowe

Poniższe parametry pozwalają na tworzenie regułek filtra bez konieczności podawania w każdej regułce wielkości pola oraz pozycji w pakiecie. Ułatwia to konstruowanie filtrów i poprawia ich czytelność.

Selektor protokołu	Selektor pola	Parametry ^a	Opis
match ip	src	<i>prefix/32</i>	adres źródłowy pakietu
	dst	<i>prefix/32</i>	adres docelowy pakietu
	tos		
	dsfield	<i>tos u8</i>	pole TOS
	precedence		
	ihl	<i>ihl u8</i>	długość nagłówka pakietu
	protocol	<i>prot u8</i>	numer protokołu enkapsulowanego
	nofrag		nie jest fragmentowany
	firstfrag		zawiera pierwszy fragment pakietu
	df		ustawiona flaga „nie fragmentować”
	mf		ustawiona flaga „więcej fragmentów”
	sport	<i>port u16</i>	port źródłowy prot. wyższej warstwy
match ip6 ^b	dport	<i>port u16</i>	port docelowy prot. wyższej warstwy
	icmp_type	<i>typ u8</i>	typ komunikatu ICMP
	icmp_code	<i>kod u8</i>	kod komunikatu ICMP
udp, tcp	src	<i>prefix/128</i>	adres źródłowy pakietu
	dst	<i>prefix/128</i>	adres docelowy pakietu
	flowlabel	<i>flow u32</i>	identyfikator przepływu IPv6
udp, tcp	src	<i>src u16</i>	port źródłowy
	dst	<i>dst u16</i>	port docelowy
icmp	type	<i>typ u8</i>	typ komunikatu ICMP
	code	<i>kod u8</i>	kod komunikatu ICMP

^aParametry podajemy w postaci liczb dziesiętnych, szesnastkowych lub adresów IPv4 i IPv6, tam gdzie to jest wymagane. Jeśli selektor przyjmuje dwa parametry, to pierwszy z nich określa wzorzec, a drugi maskę o podanej długości w bitach. Długość wzorca musi być taka sama jak maski.

^bPozostałe parametry są identyczne jak dla IPv4: `priority`, `protocol`, `dport`, `sport`, `icmp_type`, `icmp_code`

Przykład:

```
$ tc filter add dev ppp0 parent 1:0 prio 10 u32 \
    match ip tos 0x10 0xff \
    flowid 1:4
```

Powyższa regułka pasuje do pakietów, które mają pole TOS ustawione na 0x10. Pole TOS zaczyna się po pierwszym bajcie nagłówka IP i ma wielkość jednego bajtu, więc za pomocą parametrów ogólnych można tę regułkę zapisać tak:

```
$ tc filter add dev ppp0 parent 1:0 prio 10 u32 \
    match u8 0x10 0xff at 1
    flowid 1:4
```

Trzeba pamiętać, że parametry szczegółowe są zawsze tłumaczone na parametry ogólne, i w takiej postaci przechowywane w jądrze. Oznacza to ni mniej ni więcej to, że parametry `tcp src` oraz `udp src` są jednoznaczne, co wynika z faktu, że zarówno w nagłówku TCP jak i UDP numer portu źródłowego jest 16-bitowy i znajduje się na początku pakietu. Jak uniknąć takiej dwuznaczności? Filtr `u32` może posiadać kilka selektorów, połączonych w domyśle logicznym „i”:

```
tc filter add dev ppp0 parent 1:0 prio 10 u32 \
match tcp dst 22 0xffff \
    match ip protocol 0x6 0xff \
    flowid 1:2
```

11 Inne implementacje

Linuxowa implementacja algorytmów QoS nie jest, jak się łatwo domyślić, jedyna w świecie ruterów. Wiele innych systemów posiada zaimplementowane różne algorytmy kolejujące oraz mechanizmy QoS.

Cisco IOS posiada implementację algorytmów WFQ (*Weighted Fair Queueing*), PQ (*Priority Queueing*), WRED (*Weighted Random Early Detection*) oraz CBQ, przy czym to ostatnie określane jest jako `custom queueing`. IOS posiada także możliwość przycinania pasma (*traffic-shaping*) dostępnego dla danych, pasujących do określonych regułek [9].

Dla rodziny systemów BSD (FreeBSD, OpenBSD oraz NetBSD) jest dostępny pakiet ALTQ [10], zawierający implementacje algorytmów CBQ, RED, WFQ i innych.

Na stronach obu wymienionych implementacji można znaleźć bardzo dużo ogólnych informacji na temat działania algorytmów QoS, przydatnych także dla użytkowników Linuxa.

Bibliografia

- [1] B. Braden *et al.*, „Recommendations on Queue Management and Congestion Avoidance in the Internet”, kwiecień 1998 (RFC 2309)
- [2] S. Floyd, V. Jacobson, „Link-sharing and Resource Management Models for Packet Networks”, IEEE/ACM Transactions on Networking, Vol. 3 No. 4, sierpień 1995; także <http://www-nrg.ee.lbl.gov/papers/link.pdf>
- [3] S. Floyd, „Notes on Class-Based Queueing: Setting Parameters”, luty 1996, <ftp://ftp.ee.lbl.gov/papers/params.ps.Z>
- [4] S. Shenker, J. Wroclawski, „General Characterization Parameters for Integrated Service Network Elements”, wrzesień 1997 (RFC 2215)
- [5] The „Test TCP” program. Useful for network performance testing with both TCP and UDP. Also useful for setting up network pipes between machines. Originally written at ARL (then BRL). <ftp://ftp.arl.mil/pub/ttcp/>
- [6] P. Almqvist, „Type of Service in the Internet Protocol Suite”, lipiec 1992 (RFC 1349)
- [7] W. R. Stevens, „TCP/IP Illustrated, Vol. I”, Addison-Wesley 1994
- [8] K. Ramakrishnan, S. Floyd, „A Proposal to add Explicit Congestion Notification (ECN) to IP”, styczeń 1999 (RFC 2481)
(linuxowa implementacja ECN została stworzona przez Toma Kelly tom@lyndewode.co.uk i jest dostępna pod adresem <http://www.lynde.demon.co.uk/tom/ecn/ecn1.tgz>)
- [9] Cisco Systems Inc., „IOS Quality of Service”, <http://www.cisco.com/warp/public/732/Tech/quality.shtml>
- [10] Kenjiro Cho, „ALTQ: Alternate Queueing for FreeBSD”
<http://www.csl.sony.co.jp/person/kjc/programs.html>
- [11] Aleksiej Kuźniecowa, pakiet `iproute2` zawierający programy `ip` oraz `tc`, <ftp://sunsite.icm.edu.pl/pub/Linux/iproute/>
- [12] Sarawanan Radakrisznan, „Linux – Advanced Networking Overview”, <http://qos.ittc.ukans.edu/howto/index.html>
- [13] Martijn van Oosterhout, „Linux 2.2 Packet Shaping HOWTO”, <http://cupid.suninternet.com/ kleptog/Packet-Shaping-HOWTO.html>
- [14] Bert Hubert, Greg Maxwell, „Linux 2.4 Routing HOWTO”, <http://www.ds9a.nl/2.4Routing/>

Copyright 1999 by Paweł Krawczyk < *kravietz@ceti.pl* >

Warunki dystrybucji

Kopiowanie w formie elektronicznej dozwolone wyłącznie w niezmienionej postaci, z zachowaniem informacji o autorze oraz warunkach dystrybucji i w celach niekomercyjnych. Przedruk oraz sprzedaż dozwolone wyłącznie za pisemną zgodą autora.

UWAGA: obecna wersja ma jeszcze masę błędów, niedoróbek i nieścisłości, więc proszę czytać ją z krytycznym nastawieniem i nie wierzyć we wszystko co napisałem.

Uwagi, poprawki i rozszerzenia mile widziane.