Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Grzegorz Tomasz Chimosz
Nr albumu: 235944

Zaawansowana kontrola i kształtowanie ruchu sieciowego w Linuksie

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki

Wrzesień 2010

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora pracy

Streszczenie

W pracy przedstawiono projekt aplikacji służącej do kontrolowania (ograniczania) ruchu sieciowego dla poszczególnych połączeń w systemie operacyjnym Linux. Wykorzystano dyscypliny kolejkowania – część podsystemu sieciowego Linuksa, odpowiedzialną za zarządzanie przepływem pakietów przez interfejsy sieciowe. Na potrzeby aplikacji opracowane zostały dwie dyscypliny kolejkowania – dla ruchu przychodzącego i wychodzącego.

Cechą wyróżniającą stworzony program jest adresowanie go do użytkowników domowych, dla których zawiłości konfiguracji przepływu ruchu sieciowego mogą pozostać tajemnicą. Jednym z ważniejszych założeń była możliwość dokonywania dynamicznych zmian ustawień już nawiązanych połączeń. Jest to według wiedzy autora pierwsze takie rozwiązanie dla systemu operacyjnego Linux.

W pracy opisano projekt, implementację oraz wyniki testów opracowanego programu. Ponadto przedstawiono sposoby komunikacji pomiędzy przestrzenią jądra i użytkownika w Linuksie.

Słowa kluczowe

Linux, kształtowanie ruchu sieciowego, dyscyplina kolejkowania, rutowanie, sieci komputerowe, stos sieciowy

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka, nauki komputerowe

Klasyfikacja tematyczna

C. Computer Systems Organization
C.2 COMPUTER-COMMUNICATION NETWORKS
C.2.3 Network Operations
C.2.6 Internetworking
D.4 OPERATING SYSTEMS
D.4.4 Communications Management

Tytuł pracy w języku angielskim

Advanced traffic control and bandwidth management in Linux

Spis treści

Wprowadzenie 7

1. Opis podsystemu sieciowego Linuksa 9
1.1. Modele odniesienia ISO OSI oraz TCP/IP 9
1.2. Protokoły sieciowe 9
1.2.1. Protokół internetowy 10
1.2.2. Protokół kontroli transmisji 12
1.2.3. Datagramowy protokół użytkownika 13
1.3. Ruch przychodzący i wychodzący 13
1.4. Przepływ pakietów sieciowych 14
1.5. Kontrola i kształtowanie ruchu sieciowego 15
1.6. Dyscypliny kolejkowania 15
1.6.1. Bezklasowe dyscypliny kolejkowania 16
1.6.2. Filtr kubełka żetonów 16
1.6.3. Dyscypliny kolejkowania z klasami 17
1.6.4. Dyscyplina kolejkowania dla ruchu przychodzącego 17
1.6.5. Struktura opisująca dyscyplinę kolejkowania – `Qdisc_ops` 17
1.7. Klasyfikatory (filtry) i klasy 19
1.8. Najważniejsze struktury danych związane z podsystemem sieciowym Linuksa 19
1.8.1. Struktury `socket` i `sock` 19
1.8.2. Struktura `skbuff` 20
1.9. Narzędzia przestrzeni użytkownika 20
1.9.1. `tc` 21
1.10. Podsumowanie 21

2. Projekt FireQOS 23
2.1. Założenia projektowe 23
2.2. Architektura 23
2.3. Urządzenie znakowe `fireqosdev` 24
2.4. Dyscyplina kolejkowania dla ruchu przychodzącego 25
2.5. Dyscyplina kolejkowania dla ruchu wychodzącego 27
2.6. Graficzny interfejs użytkownika 27
2.7. Komunikacja pomiędzy przestrzeniami jądra i użytkownika 28
2.7.1. Urządzenia znakowe 28
2.7.2. Pomocnicze wywołania przestrzeni użytkownika 29
2.7.3. Kolejki komunikatów Systemu V 29
2.7.4. Netlink 29
2.8. Możliwe rozwiązania alternatywne i ulepszenia 30

2.8.1. Pole typu usługi w nagłówku protokołu IP 30
2.8.2. Monitorowanie gniazd i połączeń 30
2.8.3. Haki w wywołaniach systemowych 30
2.8.4. Filtr pakietów 31
2.9. Podsumowanie 31

3. Implementacja FireQOS 33
3.1. Urządzenie znakowe `fireqosdev` 33
3.2. Dyscyplina kolejkowania dla ruchu przychodzącego 33
3.3. Dyscyplina kolejkowania dla ruchu wychodzącego 35
3.4. Graficzny interfejs użytkownika 36
3.4.1. Model-Widok-Kontroler 37
3.4.2. Zadania wykonywane w tle 37
3.4.3. Sygnały i gniazda 37
3.5. Przesyłanie limitów do dyscyplin kolejkowania 38
3.6. Sposób użycia FireQOS 39
3.7. Podsumowanie 39

4. Testy 41
4.1. Opis środowiska testowego 41
4.2. Wyniki testów 41
4.2.1. Testy ograniczania pasma 41
4.2.2. Testy przepustowości dyscypliny 42
4.3. Testy zmiany ograniczenia pasma w czasie 43
4.4. Analiza i interpretacja wyników 43

5. Podsumowanie 47

A. Zawartość płyty dołączonej do pracy 49

Bibliografia 51

Spis rysunków

1.1. Modele odniesienia ISO OSI oraz TCP/IP [źródło: [Sieci], s. 53] 10
1.2. Nagłówek protokołu IP w wersji 4 [źródło: RFC 791] 10
1.3. Nagłówek protokołu IP w wersji 6 [źródło: RFC 2460] 11
1.4. Nagłówek protokołu TCP [źródło: RFC 793] 12
1.5. Nagłówek protokołu UDP [źródło: RFC 768] 13
1.6. Przepływ pakietów sieciowych [źródło: [LARTC]] 15
1.7. Cykl życia struktur powiązanych z gniazdem sieciowym 20

2.1. Architektura i komunikacja elementów FireQOS 24

3.1. Diagram klas graficznego interfejsu użytkownika FireQOS 36
3.2. Graficzny interfejs użytkownika FireQOS 37

4.1. Wynik dla limitu 50 KiB/s 42
4.2. Wynik dla limitu 500 KiB/s 42
4.3. Wynik dla limitu 5000 KiB/s 42
4.4. Porównanie dyscyplin dla ruchu przychodzącego 43
4.5. Porównanie dyscyplin dla ruchu wychodzącego 43
4.6. Testy zmiany ograniczenia pasma w czasie dla ruchu przychodzącego 44
4.7. Testy zmiany ograniczenia pasma w czasie dla ruchu wychodzącego 44

Wprowadzenie

System operacyjny Linux od wersji 2.2 posiada wbudowane mechanizmy kontroli ruchu sieciowego. Można je konfigurować i dostrajać do swoich potrzeb za pomocą kolekcji narzędzi *iproute2*[1]. Możliwe jest m.in. kształtowanie ruchu sieciowego, budowanie ścian ogniowych oraz tworzenie bram sieciowych. Ze względu na poziom skomplikowania i liczne zawiłości są to jednak rozwiązania adresowane do zaawansowanych administratorów i zostały stworzone z myślą o zastosowaniach serwerowych. Początkujący użytkownicy są skazani na proste graficzne interfejsy, które jedynie usuwają niedogodność korzystania z narzędzi konsolowych. Dzięki nim użytkownik komputera nie musi pamiętać nazw i kolejności wszystkich poleceń oraz argumentów, wciąż jednak wymagają one dużej wiedzy na temat konfigurowanych mechanizmów. Ich wadą jest również statyczność – wiele z nich zmianę konfiguracji przeprowadza przez usunięcie wszystkich reguł oraz wprowadzenie nowych ustawień. W przeciwieństwie do typowej zapory ogniowej dla systemu MS Windows nie można zablokować aplikacji dostępu do sieci w momencie, gdy próbuje ona ten dostęp uzyskać, nie można zmienić jej ograniczeń w trakcie działania. Bardzo trudne i skomplikowane jest ograniczenie ruchu dla wybranego programu korzystającego z sieci lub konkretnego połączenia (gniazda sieciowego).

Z drugiej strony wiele prób stworzenia wygodniejszych i przyjaźniejszych dla użytkownika narzędzi pokazuje zapotrzebowanie na tego typu aplikacje. Jedną ze zmian wprowadzanych do polskiego remiksu Ubuntu jest na przykład dodanie *gufw* (graficznego konfiguratora zapory ogniowej)[2].

Poszukując w sieci narzędzi o takich zastosowaniach znalazłem tylko dwa programy z graficznym interfejsem pozwalające kontrolować przydział pasma sieciowego dla poszczególnych aplikacji – *cFosSpeed*[3] oraz *NetLimiter*[4], oba komercyjne i dostępne jedynie dla MS Windows.

Stąd pomysł stworzenia programu z graficznym interfejsem służącego do kontrolowania dostępu poszczególnych aplikacji do sieci, przede wszystkim przydziału pasma w trakcie ich działania. Z założenia ma to być rozwiązanie dla użytkowników domowych, niezbyt zaawansowanych w obsłudze komputera oraz przechodzących na Linuksa z systemu MS Windows, którym brak takiego programu może wydawać się brakiem GNU/Linuksa. Umożliwienie zmiany przydziału pasma sieciowego ręcznie przez użytkownika w trakcie działania programu pozwala przede wszystkim ograniczyć ruch monopolizujący połączenie sieciowe (np. ściąganie dużych plików) i tym samym bardzo utrudniającym korzystanie z innych programów i usług internetowych.

[1] `http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2`
[2] `http://czytelnia.ubuntu.pl/index.php/2009/05/01/jurny-jarzabek-polski-remix-ubuntu-904/`
[3] `http://www.cfos.de/speed/cfosspeed.htm`
[4] `http://www.netlimiter.com`

Struktura pracy

W rozdziale 1 przedstawione zostały mechanizmy kontroli przepływu pakietów – dyscypliny kolejkowania i powiązane z nimi zagadnienia.

Rozdział 2 zawiera opis zaprojektowanego rozwiązania i jego głównych elementów oraz pomysł alternatywnego podejścia. Omówione w nim zostały również mechanizmy komunikacji pomiędzy przestrzenią jądra i użytkownika.

Rozdział 3 to opis implementacji projektu z rozdziału 2 dla Linuksa w wersji 2.6.33.4 oraz sposobu uruchomienia stworzonej aplikacji. Dodatkową wartość stanowi opis funkcji jądra pozwalających uzyskać szczegółowe informacje o odebranym pakiecie danych, standardowo niedostępnych dla dyscyplin kolejkowania ruchu przychodzącego.

Rozdział 4 zawiera wyniki testów opracowanego programu.

W rozdziale 5 podsumowuję efekty pracy.

Rozdział 1

Opis podsystemu sieciowego Linuksa

W niniejszym rozdziale przedstawię podstawowe informacje o sieciach TCP/IP i używanych w nich protokołach oraz o przepływie danych w podsystemie sieciowym Linuksa, z punktu widzenia mechanizmów kontroli ruchu sieciowego. Jego treść jest niezbędna do zrozumienia projektu i opisu implementacji programu stworzonego w ramach pracy.

1.1. Modele odniesienia ISO OSI oraz TCP/IP

Model odniesienia ISO OSI to siedmiowarstwowy model opisujący komunikację sieciową. Każda z warstw wprowadza abstrakcję i posiada dobrze określoną funkcję. Zależności pomiędzy warstwami są ograniczone do minimum, co ułatwia implementację stosu sieciowego i czyni ją wydajniejszą. Zgodnie z modelem dane – pakiety z warstw wyższych są kapsułkowane w protokołach niższych warstw.

Istnieje również uproszczony model odniesienia TCP/IP, który skleja niektóre z warstw modelu ISO OSI. Przedstawia to rysunek 1.1.

Model TCP/IP wskazuje również konkretne protokoły sieciowe omówione w rozdziale 1.2.

Linuksowa implementacja stosu TCP/IP jest w dużym stopniu zgodna z modelem, w szczególności zachowana jest zasada niezależności pomiędzy protokołami różnych warstw.

Każdy z obsługiwanych protokołów zgłasza strukturę z adresami funkcji do obsługi pakietów go wykorzystujących. Dla protokołów warstwy sieciowej jest to struktura `packet_type`[1], dla warstwy transportowej – struktura `net_protocol`[2]. Wyszukiwanie tych struktur w kodzie jądra pozwala łatwo znaleźć implementację wszystkich dostępnych protokołów.

Warto pamiętać, że przekazywanie pakietu danych do wyższych warstw stosu sieciowego wiąże się z usuwaniem nagłówków warstw niższych. Z tego powodu nie można korzystać z niektórych funkcji udostępnianych przez jądro Linuksa, co zostanie szerzej omówione w opisie implementacji dyscypliny kolejkowania dla ruchu przychodzącego (patrz rozdział 3.2).

1.2. Protokoły sieciowe

Ponieważ celem niniejszej pracy jest opracowanie aplikacji pozwalającej kontrolować ruch i przydział pasma sieciowego, a niemal wszystkie sieci komputerowe, w tym największa –

[1] `include/linux/netdevice.h#L1125`
[2] `include/net/protocol.h#L36`

Model ISO OSI	Model TCP/IP
7: Warstwa aplikacji	4: Warstwa aplikacji
6: Warstwa prezentacji	*(nie występuje)*
5: Warstwa sesji	
4: Warstwa transportowa	3: Warstwa transportowa
3: Warstwa sieciowa	2: Warstwa internetowa
2: Warstwa łącza danych	1: Warstwa host-sieć
1: Warstwa fizyczna	

Rysunek 1.1: Modele odniesienia ISO OSI oraz TCP/IP [źródło: [Sieci], s. 53]

0	4	8	16	19	24–32
Wersja	IHL	Typ usługi	Długość		
Identyfikator			Flagi	Przesunięcie framgmentu	
TTL		Protokół	Suma kontrolna		
Adres źródłowy					
Adres docelowy					
Opcje					Wyrównanie

Rysunek 1.2: Nagłówek protokołu IP w wersji 4 [źródło: RFC 791]

Internet, są zgodne z modelem TCP/IP skupiam się na wykorzystywanych w nich protokołach. W szczególności podstawowe informacje o ich budowie są niezbędne przy implementacji dyscypliny kolejkowania (patrz rozdział 1.6) dla ruchu przychodzącego.

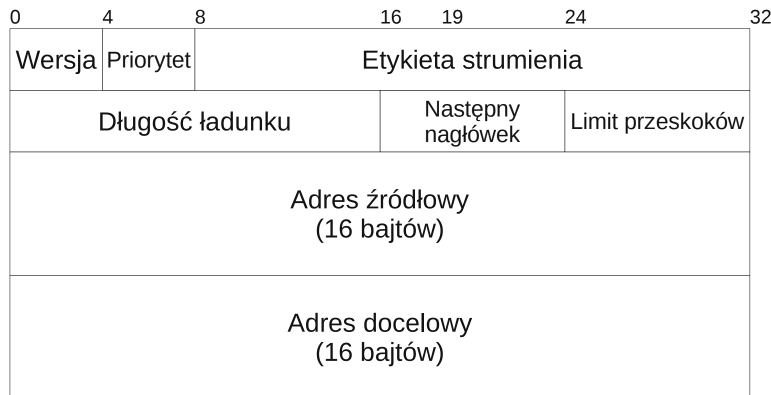
1.2.1. Protokół internetowy

Protokół internetowy, nazywany w skrócie IP (od ang. *Internet Protocol*), jest bezstanowy, bezpołączeniowy, nie gwarantuje dostarczenia danych do odbiorcy, nie zapewnia kolejności odebrania zgodnej z kolejnością nadawania pakietów. Ponadto możliwe jest zduplikowanie pakietów danych.

Nagłówek powszechnie używanej wersji 4 tego protokołu został przedstawiony na rysunku 1.2.

Najistotniejsze pola z punktu widzenia niniejszej pracy to:

- **Wersja** – pozwala sprawdzić, czy nagłówek jest w wersji 4,
- **Długość nagłówka (ang. *Internet Header Length*)** – długość nagłówka



Rysunek 1.3: Nagłówek protokołu IP w wersji 6 [źródło: RFC 2460]

w 32-bitowych słowach, wskazuje początek danych zawartych w pakiecie (początek nagłówka protokołu wyższego poziomu, np. TCP),

- **Długość całkowita** – długość całego pakietu (nagłówka oraz danych),
- **Typ usługi (ang. *Type of Service*)** – jest opisany dalej,
- **Protokół** – identyfikator protokołu wyższego poziomu (m.in. TCP, UDP),
- **Suma kontrolna** – pozwala zweryfikować poprawność pakietu,
- **Adres źródłowy i docelowy** – 32-bitowe adresy nadawcy i odbiorcy pakietu.

Pole określające typ usługi zostało wprowadzone, by określić ważność pakietu i preferencje nadawcy. Trzy pierwsze bity tego pola specyfikują ważność, np. zwykłe dane lub sterowanie siecią. Pozostałe służą do wskazania co jest najważniejsze dla nadawcy – niskie opóźnienia (np. sesje ssh), wysoka przepustowość (np. transmisje video), wysoka niezawodność połączenia (mała liczba gubionych pakietów), czy minimalizacja kosztu. Dokładny opis możliwych wartości pola ToS znajduje się w RFC 1349.

Strukturą opisującą nagłówek IPv4 w Linuksie jest `iphdr`[3].

Od wielu lat trwają prace nad wdrożeniem szóstej wersji protokołu IP, jednak jego popularność jest niewielka i trudno przewidywać, kiedy IPv6 stanie się podstawowym obowiązującym protokołem warstwy sieciowej w Internecie i innych sieciach.

Nagłówek protokołu IP w wersji 6 przedstawia rysunek 1.3.

Trzeba pamiętać, że Linuksowa implementacja IPv6 zawiera zaszłość historyczną, która może prowadzić do napisania błędnego kodu. Struktura `ipv6hdr`[4] opisuje nagłówek w wersji przedstawionej w nieaktualnym już RFC 1883 (zastąpionym przez RFC 2460). Różnica polega na użyciu 4 pierwszych bitów pola **Etykieta przepływu** do rozszerzenia pola **Klasa ruchu** (wcześniej **Priorytet**).

Najistotniejsze pola z punktu widzenia niniejszej pracy to:

- **Wersja** – pozwala sprawdzić, czy nagłówek jest w wersji 6,

[3] `include/linux/ip.h#L85`
[4] `include/linux/ipv6.h#L107`

Rysunek 1.4: Nagłówek protokołu TCP [źródło: RFC 793]

- **Klasa ruchu** – odpowiednik pola **Typ usługi** z IPv4; w chwili pisania pracy są przypisane jedynie wartości testowe – do eksperymentów, szczegóły można znaleźć w RFC 2460, rozdział 7 oraz w RFC 4727, rozdział 3.2,

- **Długość ładunku** – długość danych pakietu (z pominięciem długości nagłówka w przeciwieństwie do IPv4),

- **Następny nagłówek** – wskazuje początek kolejnego nagłówka; w szczególności specyfikuje, czy po bieżącym nagłówku jest nagłówek TCP lub UDP,

- **Adres źródłowy i docelowy** – 16-bajtowe adresy nadawcy i odbiorcy pakietu.

Dodatkowego komentarza wymaga pole **Następny nagłówek**. Ponieważ nagłówek IPv6 jest uproszczony w stosunku do nagłówka IPv4, stworzono mechanizm pozwalający umieścić kilka nagłówków z istniejących sześciu typów (patrz [Sieci], s. 403). Po ostatnim nagłówku IPv6 zaczyna się nagłówek protokołu wyższego poziomu (np. TCP lub UDP).

1.2.2. Protokół kontroli transmisji

Protokół kontroli transmisji, w skrócie TCP (od ang. *Transmission Control Protocol*), jest najpowszechniej stosowanym protokołem warstwy transportowej. Zapewnia on dwukierunkowe połączenie strumieniowe pomiędzy dwoma węzłami w sieci. Jest odporny na zaginięcie pakietów danych w sieci, zmianę ich kolejności oraz duplikaty. Wykorzystują go prawie wszystkie usługi w Internecie – strony WWW, protokoły POP3, SMTP do obsługi poczty, ssh oraz wiele innych.

Nagłówek protokołu TCP został przedstawiony na rysunku 1.4.

Najistotniejsze pola z punktu widzenia niniejszej pracy to:

- **Port źródłowy i docelowy** – 16-bitowe numery portów źródłowego i docelowego,

- **Suma kontrolna** – pozwala zweryfikować poprawność pakietu.

Rysunek 1.5: Nagłówek protokołu UDP [źródło: RFC 768]

Implementacje TCP zawierają algorytm powolnego startu, który bazując na częstotliwości przesyłanych potwierdzeń określa częstotliwość wysyłania kolejnych pakietów, tak by maksymalnie wykorzystać dostępną przepustowość. W połączeniu z algorytmem zapobiegania zatorom umożliwia dostosowanie szybkości wysyłania do możliwości łącza i odbiorcy. Znaczenie tego mechanizmu zostanie omówione szerzej w rozdziale 1.3.

Linuksowa implementacja TCP jest podzielona względem wykorzystywanej wersji protokołu warstwy sieciowej; dla IPv4 znajduje się ona w katalogu `/net/ipv4`, dla IPv6 – w `/net/ipv6`. Strukturą opisującą nagłówek TCP w Linuksie jest `tcphdr`[5].

1.2.3. Datagramowy protokół użytkownika

Datagramowy protokół użytkownika, w skrócie UDP (od ang. *User Datagram Protocol*), to prosty bezstanowy protokół, którego główną zaletą jest niewielki narzut. Nie zapewnia gwarancji dostarczenia ani zachowania kolejności wysyłanych pakietów, jednak pozwala na realizację usług wymagających małych opóźnień, np. strumieniowej transmisji video.

Nagłówek protokołu UDP został przedstawiony na rysunku 1.5.

Najistotniejsze pola z punktu widzenia niniejszej pracy to:

- **Port źródłowy i docelowy** – 16-bitowe numery portów źródłowego i docelowego,
- **Suma kontrolna** – pozwala zweryfikować poprawność pakietu.

Strukturą opisującą nagłówek UDP w Linuksie jest `udphdr`[6].

1.3. Ruch przychodzący i wychodzący

Istnieje fundamentalna różnica pomiędzy ruchem przychodzącym a wychodzącym – pierwszego, pomijając mechanizm adaptacji w protokole TCP (patrz rozdział 1.2.2), nie można kontrolować. Interfejs sieciowy odbiera pakiety, nie mając wpływu na ich kolejność, rozmiar ani czas dostarczenia. Ogranicza to możliwość kształtowania ruchu przychodzącego do odrzucania odebranych ramek (ang. *policing*) oraz akceptowania (np. dostarczania do aplikacji lub rutowania pakietu dalej). Opóźnienie dostarczenia pakietu nie jest rozsądnym rozwiązaniem – przepustowość łącza została wykorzystana; jeśli podjęto decyzję o zaakceptowaniu pakietu, to aplikacja kiedyś musi ten pakiet odebrać i przetworzyć, warto więc dostarczyć go jak najszybciej, w celu lepszego wykorzystania czasu procesora. Komentarza wymaga sytuacja,

[5] `include/linux/tcp.h#L24`
[6] `include/linux/udp.h#L22`

gdy pakiet jest tylko rutowany. Zaakceptowanie pakietu przez dyscyplinę kolejkowania ruchu przychodzącego (patrz rozdział 1.6.4) nie oznacza jego natychmiastowego wysłania; trafi on do kolejki ruchu wychodzącego dla danego interfejsu, o ile taka jest skonfigurowana.

Obejściem tego ograniczenia są rozwiązania tworzące wirtualny interfejs sieciowy, do którego przekazywany jest ruch przychodzący do interfejsu fizycznego. Najpopularniejszym jest *Intermediate Queueing Device*[7]. Pozwala ono wykorzystać dostępne dyscypliny kolejkowania dla ruchu wychodzącego (ang. *egress*) w miejscu dyscyplin dla ruchu przychodzącego (ang. *ingress*), co jest o tyle istotne, że tych pierwszych jest znacznie więcej i oferują większe możliwości. Jednakże pomimo kilku lat historii projektu, sięgającej czasów Linuksa 2.4, nie został on włączony do głównej gałęzi jądra, jako niezgodny z koncepcją podsystemu sieciowego odpowiadających za niego osób[8]. Postulują oni przeniesienie określania polityki (ang. *policing*) do poziomu sterownika sieciowego `dummy`, co wydaje się być dosyć eleganckie – skoro odrzucanie pakietu wygląda tak, jakby pakiet w ogóle nie został odebrany, to rzeczywiście jest to właściwe miejsce. Jednakże takie podejście ogranicza jeszcze bardziej i tak skromne możliwości dyscyplin kolejkowania ruchu przychodzącego.

Właściwie należałoby zgodzić się z opinią, że próby kontrolowania ruchu przychodzącego to rozwiązywanie problemu po niewłaściwej stronie łącza. Jednak przeciętny użytkownik komputera nie ma żadnego wpływu na kształtowanie ruchu po stronie swojego dostawcy, a doświadczenie pokazuje, że nie stosują oni nawet najprostszych mechanizmów pozwalających sprawiedliwie dzielić pasmo pomiędzy przechodzące strumienie połączeń TCP, nie wspominając o bardziej zaawansowanych metodach, np. priorytyzacji pakietów na podstawie pola określającego typ usługi w nagłówku IPv4. Takie podejście tłumaczy koszt obliczeniowy związany z zaawansowaną kontrolą przepływu. Rezygnacja z niej sprawia, że ruter dostawcy może ograniczyć analizę nagłówków, w szczególności nie musi sprawdzać zawartości wszystkich pól nagłówka IP, nie wspominając o sięganiu do nagłówka protokołu warstwy transportowej.

Dla ruchu wychodzącego możliwości manipulacji są znacznie większe. W [LARTC] (rozdział 9.4) wyróżniono następujące związane z nim definicje:

- **planowanie** (ang. *scheduling* oraz *reordering*) – zmiana kolejności pakietów, często podejmowana przy użyciu klasyfikatorów; pozwala m.in. nadać większy priorytet transmisjom wymagającym interaktywności, np. sesji `ssh`,

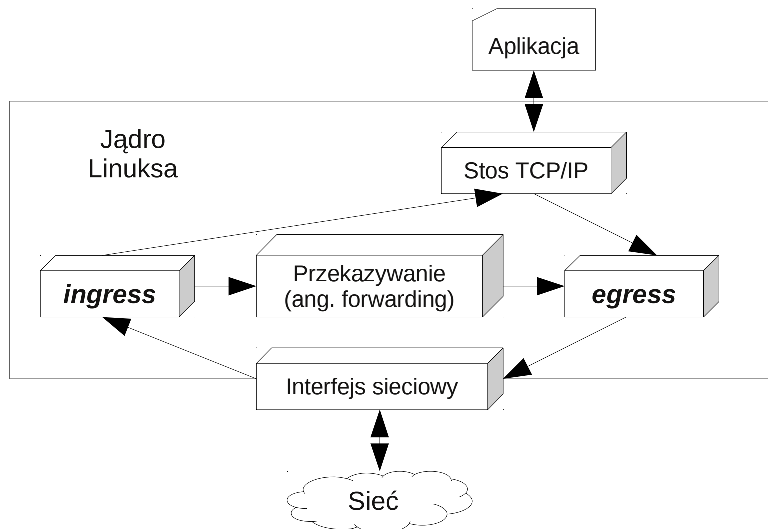
- **kształtowanie** (ang. *shaping*) opóźnianie wysyłki pakietów sieciowych w celu ograniczenia wykorzystania pasma; zalecane jest odróżnianie kształtowania od określania polityki, które jedynie odrzuca przychodzące pakiety.

Te różnice znajdują swoje odzwierciedlenie w podsystemie sieciowym Linuksa. API wykorzystywane przez dyscypliny dla ruchu przychodzącego i wychodzącego tylko pozornie wygląda tak samo, zostanie to szerzej omówione w rozdziale 1.6.4.

1.4. Przepływ pakietów sieciowych

Rysunek 1.6 obrazuje przepływ pakietów sieciowych z zaznaczeniem dyscyplin kolejkowania dla ruchu przychodzącego i wychodzącego.

[7] `http://www.linuximq.net`
[8] `http://wiki.nix.hu/cgi-bin/twiki/view/IMQ/ImqFaq#Why_IMQ_is_not_in_the_official_k`



Rysunek 1.6: Przepływ pakietów sieciowych [źródło: [LARTC]]

1.5. Kontrola i kształtowanie ruchu sieciowego

Z punktu widzenia kształtowania ruchu sieciowego (kontroli przydziału pasma) istotne są następujące elementy infrastruktury sieciowej Linuksa:

- dyscypliny kolejkowania – patrz rozdział 1.6,
- klasyfikatory (filtry) – patrz rozdział 1.7,
- klasy.

Do konfiguracji wymienionych mechanizmów służą narzędzia wchodzące w skład pakietu *iproute2* – `tc` oraz `ip`.

1.6. Dyscypliny kolejkowania

Dyscypliny kolejkowania (ang. *queueing disciplines*) są elementem wykonawczym kontroli pasma w Linuksie. Znajdują się w pierwszej warstwie modelu TCP/IP (patrz rozdział 1.1), jednak mając dostęp do całego pakietu mogą sięgać do nagłówków protokołów wyższych poziomów. Ukrywają w sobie algorytmy planowania i kształtowania ruchu dla ruchu wychodzącego lub określania polityki dla ruchu przychodzącego (patrz rozdział 1.3).

Dyscyplina kolejkowania to usługa rejestrowana w jądrze Linuksa za pomocą funkcji `int register_qdisc(struct Qdisc_ops *qops)`. Typowo cała funkcjonalność jest realizowana jako moduł Linuksa. Od momentu zgłoszenia, instancja dyscypliny może zostać podłączona do interfejsu sieciowego jako dyscyplina *egress*, czyli dla ruchu wychodzącego lub jako dyscyplina *ingress* – dla ruchu przychodzącego. Eksperymenty pokazują, że tylko dyscyplina o nazwie `ingress` (pole `id` struktury `Qdisc_ops`) może zostać skonfigurowana w charakterze dyscypliny *ingress*, ponadto nie może ona pełnić funkcji dyscypliny *egress*. Z tego powodu implementację własnej dyscypliny dla ruchu przychodzącego warto nazwać `ingress`, aby uniknąć kłopotliwej modyfikacji jądra Linuksa i narzędzia `tc`.

Dyscyplina kolejkowania dla ruchu wychodzącego pełni funkcję kolejki, która jest informowana o nowym pakiecie (zbudowanym w wyniku zapisu do gniazda sieciowego lub odebrania pakietu rutowanego) w postaci wskaźnika do struktury `skbuff` (patrz rozdział 1.8.2). Podejmuje wtedy decyzję o dodaniu go do wewnętrznej kolejki lub, w przypadku braku miejsca, odrzuceniu pakietu. W momencie, kiedy interfejs sieciowy jest gotowy do wysyłania danych, wywoływana jest funkcja pobrania wskaźnika do struktury `skbuff`. Jeżeli pakiet jest dostępny, to dyscyplina przekazuje wskaźnik i usuwa pakiet z wewnętrznej kolejki. W przypadku dyscypliny stosującej kształtowanie możliwe jest odroczenie wykonania żądania za pomocą dostępnego dla dyscyplin mechanizmu zegarowego (ang. *watchdog*). Proces ten jest dokładniej wyjaśniony w opisie Filtra kubełka żetonów (patrz rozdział 1.6.2).

Dyscypliny kolejkowania dla ruchu przychodzącego można traktować jak filtr, który przepuszcza pakiet lub go odrzuca.

Uwagę zwraca fakt, że dyscypliny kolejkowania w bardzo naturalny sposób mogą zawierać się w sobie; nawet najprostsza kolejka `fifo` nie realizuje samodzielnie funkcji kolejki prostej, lecz korzysta z mechanizmu dostępnego w API dla dyscyplin kolejkowania.

1.6.1. Bezklasowe dyscypliny kolejkowania

Dyscypliny kolejkowania zbudowane w oparciu o proste kolejki, które jedynie przyjmują i wydają pakiety, ewentualnie odraczając zwrócenie w czasie, są określane dyscyplinami bezklasowymi (ang. classless). Najlepszym przykładem jest najczęściej stosowna kolejka prosta FIFO (od ang. *First In First Out*). Dyscyplina bezklasowa może być jednak znacznie bardziej skomplikowana (czego przykładem jest dyscyplina dla ruchu wychodzącego stworzona w ramach niniejszej pracy, patrz rozdział 3.3). Kryterium bezklasowości jest nieudostępnianie mechanizmu podłączania kolejnych dyscyplin za pomocą narzędzia `tc` (patrz rozdział 1.9.1).

1.6.2. Filtr kubełka żetonów

Filtr kubełka żetonów (ang. *Token Bucket Filter*) to jedna z bezklasowych dyscyplin kolejkowania dostępnych w Linuksie.

Przeznaczeniem tej dyscypliny jest ograniczenie ruchu wychodzącego do zadanej szybkości R. W tym celu posługuje się ona pomysłem kubełka żetonów o pojemności B, który na początku jest pełny. Żetony są dodawane do kubełka wraz z upływem czasu, w liczbie zależnej od R. Wysłanie pakietu zabiera tyle żetonów, ile wynosi długość pakietu. Jeśli jest ich zbyt mało, to obliczany jest czas, potrzebny na uzupełnienie kubełka do wymaganej liczby żetonów (długości pakietu) i wykonanie przekazania pakietu jest odraczane za pomocą mechanizmu zegarowego, który po odczekaniu spróbuje ponownie wyjąć pakiet z kolejki.

Pojemność kubełka określa maksymalny możliwy do zebrania zapas żetonów. Po napełnieniu kubełka żetony przepadają. W przeciwnym przypadku długa przerwa w nadawaniu pakietów mogłaby skutkować wysyłaniem znacznie szybszym niż R.

Nierówność w zaczerpniętej z implementacji Filtra kubełka żetonów[9], definicji 1.6.1 pozwala rozwiązać problem dokładania żetonów do kubełka.

Definicja 1.6.1 *Przepływ pakietów o długościach s_j obsłużonych w momentach t_j jest zgodny z Filtrem kubełka żetonów o przepustowości (ang.* rate*) R i pojemności (ang.* depth*) B, gdy*

$$\forall_{i \leqslant k} \sum_{j=i..k} s_j \leqslant B + R \times (t_k - t_i)$$

[9] `net/sched/sch_tbf.c#L40`

Omawiana dyscyplina przy próbie pobrania z niej kolejnego pakietu oblicza różnicę aktualnego czasu i czasu wysłania ostatniego pakietu (który przechowuje). Mnożąc otrzymaną wartość przez parametr R uzyskuje liczbę nowych żetonów i dokłada je do kubełka.

Dla dokładności limitowania pasma bardzo duże znaczenie ma dokładność i rozdzielczość używanego zegara systemowego (stała `HZ` konfigurowana w jądrze Linuksa).

1.6.3. Dyscypliny kolejkowania z klasami

Dyscypliny kolejkowania mogą zawierać w sobie klasy – poddyscypliny, do których kierują przychodzące pakiety na podstawie określonych kryteriów. Do klasyfikacji pakietów z reguły wykorzystuje się filtry (patrz rozdział 1.7), wtedy możliwa jest konfiguracja za pomocą narzędzia `tc`. Alternatywnie dyscyplina może samodzielnie podejmować decyzję o przekazaniu pakietu do konkretnej klasy – poddyscypliny.

Charakterystyczną cechą konfiguracji dyscyplin z klasami jest budowanie drzewa dyscyplin. Przychodzące pakiety są kierowane do korzenia, który decyduje o dodaniu pakietu do swojej wewnętrznej kolejki lub o przekazaniu go do jednego z dzieci w drzewie dyscyplin. Proces ten powtarza się aż do dodania pakietu lub dotarcia do liścia drzewa. Korzeń jest jedynym elementem drzewa, do którego odwołuje się podsystem sieciowy, zarówno przy dodawaniu, jak i pobieraniu pakietów. Przekazywanie pakietu pomiędzy węzłami drzewa jest dzięki temu ukryte i dyscyplina może być obsługiwana dokładnie tak samo jak bezklasowa.

Najczęściej używanymi dyscyplinami z klasami są Kolejkowanie oparte o klasy (ang. *Class Based Queueing*) oraz będące szczególnym przypadkiem, nieco prostsze w budowie, Hierarchiczne kubełki żetonów (ang. *Hierarchichal Token Bucket*). Pierwsza z nich pozwala zbudować drzewo dowolnych dyscyplin, druga z kolei wykorzystuje pomysł z Filtra kubełka żetonów (patrz rozdział 1.6.2), tworząc hierarchię kubełków z możliwością przekazywania niewykorzystanych żetonów w dół drzewa.

1.6.4. Dyscyplina kolejkowania dla ruchu przychodzącego

W jądrze Linuksa dostępna jest tylko jedna dyscyplina kolejkowania *ingress* (dla ruchu przychodzącego) o tej samej nazwie. Jej implementacja sprowadza się do aplikowania kolejnych klasyfikatorów (filtrów) i zależnie od przekazanego przez nie wyniku akceptowania lub odrzucania pakietu (zgodnie ze skonfigurowaną polityką).

Problem kontroli ruchu przychodzącego został szerzej omówiony w rozdziale 1.3. Dostępność w oficjalnym jądrze Linuksa tylko jednej dyscypliny kolejkowania uzasadnia stanowisko osób odpowiedzialnych za podsystem sieciowy; `ingress` wraz z filtrami pozwala uzyskać wszystko, co jest możliwe przy takim podejściu.

1.6.5. Struktura opisująca dyscyplinę kolejkowania – `Qdisc_ops`

Struktura `Qdisc_ops` (`include/net/sch_generic.h#L107`) zawiera m.in. wskaźniki do następujących funkcji[10]:

- `int (*init)(struct Qdisc *, struct nlattr *)`
  Funkcja inicjująca instancję dyscypliny kolejkowania. Za pomocą `qdisc_priv` można uzyskać wskaźnik do zaalokowanego przez jądro obszaru na prywatne dane dyscypliny. Drugi wskaźnik służy do przekazania ustawień, podanych w wierszu komend `tc`. Skorzystanie z tej funkcjonalności wymaga modyfikacji wspomnianego narzędzia.

[10] Opis sporządzony na podstawie `http://eprints.usq.edu.au/1280/3/Braithwaite_AppendixB_qdisc.pdf` oraz `net/sched/sch_api.c`

- `void (*reset)(struct Qdisc *)`
  Funkcja przywracająca kolejkę do stanu początkowego, powinna zwolnić pamięć zajmowaną przez wszystkie zakolejkowane pakiety oraz zresetować statystyki.

- `void (*destroy)(struct Qdisc *)`
  Funkcja zwalniająca zaalokowaną pamięć oraz inne zasoby przed usunięciem instancji dyscypliny.

- `int (*change)(struct Qdisc *, struct nlattr *arg)`
  Funkcja zmieniająca konfigurację dyscypliny; podobnie do `init` wywoływana po użyciu narzędzia `tc`.

Dokładniejszego omówienia wymagają dwie funkcje, których implementacja jest obligatoryjna dla dyscyplin kolejkowania ruchu wychodzącego – `enqueue` oraz `dequeue`.

Funkcja `int (*enqueue)(struct sk_buff *, struct Qdisc *)` służy do dodawania pakietu do kolejki. Jeśli dodawanie się uda, to funkcja przekazuje wartość `NET_XMIT_SUCCESS`. Możliwe jest też odrzucenie pakietu przekazanego jako argument lub usunięcie innego pakietu już dodanego do kolejki. W takich przypadkach zwracane są następujące kody błędów:

- `NET_XMIT_DROP` – odrzucony pakiet przekazany jak argument; ponowna próba zostanie podjęta, gdy w kolejce zwolni się miejsce,

- `NET_XMIT_CN` – zaakceptowany pakiet przekazany jak argument, prawdopodobnie odrzucony inny (ang. *congestion notification* – informacja o przeciążeniu),

- `NET_XMIT_POLICED` – pakiet odrzucony przez nałożoną politykę; ponowna próba zostanie podjęta po upływie krótkiego losowego czasu (analogicznie do mechanizmu kolizji w sieciach Ethernet), dla aplikacji czasu rzeczywistego zostanie przekazana informacja o błędzie.

Funkcja `enqueue` dyscypliny dla ruchu przychodzącego nie dodaje pakietów do wewnętrznej kolejki, a jedynie przekazuje stosowną wartość.

Funkcja `struct sk_buff * (*dequeue)(struct Qdisc *)` przekazuje wskaźnik do pakietu, który powinien zostać wysłany (o ile taki istnieje). Jeżeli dyscyplina kolejkowania dysponuje pakietem, ale w danym momencie rezygnuje z wysyłania (ze względu na przekroczenie limitu transferu), to może skorzystać z dostępnego mechanizmu zegarowego, który ponowi próbę za określony czas.

Funkcja `dequeue` w dyscyplinie *ingress* nie jest w ogóle wywoływana.

Z funkcją `dequeue` jest też powiązana funkcja `peek`. Służy ona do podejrzenia następnego pakietu, jednak bez usuwania go z kolejki. Przy implementacji należy zwrócić szczególną uwagę, by wskaźnik do pakietu przekazywany przez `peek` był jednocześnie pierwszym przekazanym przez `dequeue`. Ma to znaczenie w przypadku podłączenia implementowanej dyscypliny jako klasy w dyscyplinie z klasami.

Instancja dyscypliny kolejkowania jest opisywana przez strukturę `Qdisc`[11]. Przy implementacji własnych dyscyplin należy zwrócić szczególną uwagę na statystki związane z następującymi polami:

- `struct sk_buff_head q`
  Jest to domyślna kolejka prosta, z używania której można zrezygnować. Jednak koniecznie trzeba dbać o właściwą wartość atrybutu `qlen` – liczby pakietów aktualnie oczekujących w dyscyplinie.

[11] `include/net/sch_generic.h#L38`

- `struct gnet_stats_basic_packed bstats`
  Przy dodawaniu pakietu do dyscypliny należy zwiększyć odpowiednie atrybuty:
  - `bytes` – o długość pakietu,
  - `packets` – o 1 (liczbę dodanych pakietów).
- `struct gnet_stats_queue qstats`
  Atrybut `backlog` reprezentuje liczbę bajtów aktualnie przechowywanych w dyscyplinie. W funkcji `enqueue` należy go zwiększyć o długość pakietu (`qdisc_pkt_len(skb)`), w `dequeue` – odpowiednio zmniejszyć. W przypadku odrzucenia pakietu (gdy brak miejsca w kolejce dla ruchu wychodzącego lub w dyscyplinie dla ruchu przychodzącego) należy zwiększyć atrybut `drops` o 1 (liczbę odrzuconych pakietów).

W przypadku dyscyplin kolejkowania ważną rolę odgrywają kwestie współbieżności. Jak sugeruje praca [Grzegórski] i potwierdzają moje eksperymenty, funkcje `enqueue` i `dequeue` wykonują się przy wyłączonych przerwaniach i jeśli trwają zbyt długo, to są wywłaszczane. Skutkuje to zgłaszaniem przez planistę (ang. *scheduler*) błędu szeregowania w trakcie operacji atomowej (przy wziętej blokadzie).

Ponieważ nie można zrezygnować z ochrony wewnętrznych danych dyscypliny w środowisku współbieżnym, jedynym właściwym sposobem komunikacji z nią powinny być metody `change` i `dump`, korzystające z mechanizmu Netlink (patrz rozdział 2.7.4). Funkcje te, podobnie jak wszystkie pozostałe udostępniane przez dyscyplinę, są wykonywane po jej zablokowaniu. Tym samym dodatkowa synchronizacja nie jest potrzebna (w danej chwili wykonuje się co najwyżej jedna z funkcji).

1.7. Klasyfikatory (filtry) i klasy

Dyscyplina kolejkowania z klasami potrzebuje filtrów, aby przydzielać pakiety do swoich klas. Filtr mając dostęp do pakietu (poprzez wskaźnik do struktury `skbuff` – patrz rozdział 1.8.2) może sprawdzić zawartość nagłówków, w tym adresy, numery portów, ponadto może korzystać z opcjonalnego oznakowania pakietu przez wbudowaną w jądro Linuksa ścianę ogniową.

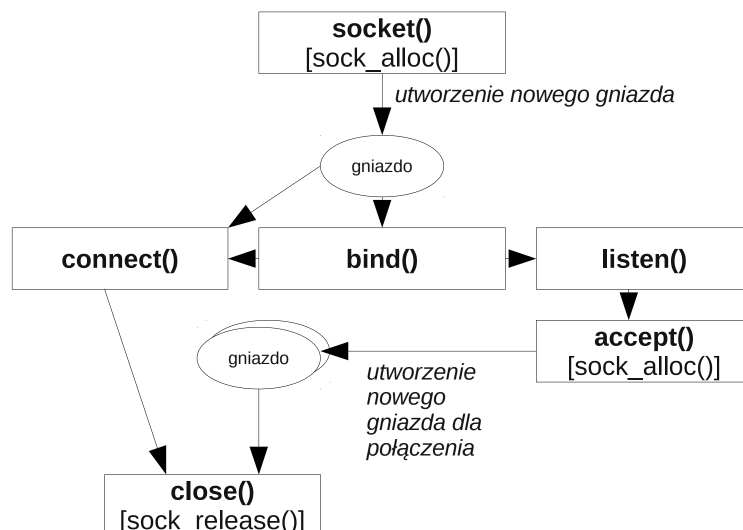
Filtry tworzą łańcuch (ang. *classifier chain*) i są kolejno przykładane do pakietu, tak długo, aż pakiet trafi do odpowiedniej klasy (bez kolejnych podklas) lub skończy się lista filtrów. Pakiety mogą być filtrowane tylko w kierunku od korzenia do liści drzewa klas.

Przykładem filtru jest zaimplementowany w jądrze Linuksa `u32`, który pozwala na efektywne dopasowywanie nagłówków pakietu. Dzięki zastosowaniu tablic mieszających pozwala na zmniejszenie kosztu obliczeniowego wykonywania kolejnych testów (gdy jest kilka instancji filtru w łańcuchu).

1.8. Najważniejsze struktury danych związane z podsystemem sieciowym Linuksa

1.8.1. Struktury `socket` i `sock`

Struktura `socket` służy do reprezentacji gniazda BSD, tj. mechanizmu komunikacji międzyprocesowej, najczęściej powiązanej z siecią komputerową. W Linuksie jest ona bardzo prosta i opisuje typ (np. `SOCK_STREAM`), stan (np. `SS_CONNECTED`) i opcje gniazda. Zawiera też wskaźniki do struktur `file` i `sock` oraz elementy związane z synchronizacją procesów.



Rysunek 1.7: Cykl życia struktur powiązanych z gniazdem sieciowym

Ze strukturą `socket` jest ściśle powiązana struktura `sock`. Przechowuje ona informacje dotyczące warstwy sieciowej dla danego gniazda. Alokowana jest jako część struktury opisującej gniazdo należące do danej rodziny protokołów, np. `inet_sock` dla `PF_INET` (patrz [Understanding], rozdział 21.1.2). Jest ona bardziej użyteczna przy identyfikowaniu gniazda w kodzie jądra – większość funkcji przyjmujących gniazdo sieciowe jako argument wymaga wskaźnika do struktury `sock`.

Rysunek 1.7 przedstawia cykl życia gniazda sieciowego w kontekście wywołań systemowych (z wyjątkiem `send()` i `recv()`).

1.8.2. Struktura `skbuff`

Struktura `skbuff` jest jedną z najważniejszych w podsystemie sieciowym. Służy do przechowywania pojedynczych pakietów, zarówno wysyłanych, jak i odbieranych. Podsystem sieciowy Linuksa przekazując pakiet pomiędzy wywołaniami różnych funkcji używa tylko i wyłącznie tej struktury.

Zawiera ona wskaźniki do właściwej treści pakietu (w tym do nagłówków używanych protokołów) oraz jego metadane (np. czas otrzymania, powiązane gniazdo sieciowe, oznakowanie wprowadzone przez kontrolę przepływu).

Pakiety przychodzące, ze względów wydajnościowych, w momencie gdy trafiają do dyscypliny kolejkowania *ingress*, nie mają określonych niektórych własności w strukturze `skbuff`, m.in. wskaźnik `struct sock * sock` jest równy `NULL`.

1.9. Narzędzia przestrzeni użytkownika

Do kontroli podsystemu sieciowego Linuksa służą narzędzia wchodzące w skład pakietu *iproute2*. Komunikację pomiędzy przestrzeniami jądra i użytkownika zapewnia Netlink API – mechanizm ściśle powiązany z mechanizmem gniazd sieciowych pozwalający przesyłać komunikaty z instrukcjami, konfiguracją oraz diagnostyczne. Najważniejszymi programami w *ipro-*

ute2 są `ip` oraz `tc`. Pierwszy z nich służy do konfiguracji m.in. interfejsów oraz rutowania. Jego celem jest zastąpienie używanych od dawna narzędzi `ipconfig`, `route`, `arp`, które jednak mają nieco prostszą składnię.

1.9.1. tc

Narzędzie `tc` (ang. *traffic control*) służy do konfiguracji dyscyplin kolejkowania oraz powiązanych z nimi filtrów. Za pomocą polecenia

```
# tc -s qdisc
```

można obejrzeć aktualną konfigurację dyscyplin wraz z podstawowymi statystykami.

Składnia pozwalająca dodawać, modyfikować ustawienia i usuwać dyscypliny wygląda następująco:

```
# tc qdisc [ add | del | replace | change | show ] dev DEVICE
       [ handle QHANDLE ] [ root | ingress | parent CLASSID ]
       [ estimator INTERVAL TIME\_CONSTANT ]
       [ stab [ help | STAB\_OPTIONS] ]
       [ [ QDISC\_KIND ] [ help | OPTIONS ] ]
```

Najistotniejsze w wywołaniu parametry to:

- `DEVICE` – nazwa urządzenia sieciowego, np. `eth0`,
- `QDISC_KIND` – nazwa dyscypliny kolejkowania,
- `OPTIONS` – niektóre z dyscyplin wymagają określenia wartości swoich parametrów,
- `QHANDLE` oraz `CLASSID` – w przypadku dyscyplin z klasami pozwala wskazywać, o który węzeł chodzi.

1.10. Podsumowanie

W tym rozdziale przedstawiłem część podsystemu sieciowego Linuksa odpowiedzialną za kontrolę przepływu – dyscypliny kolejkowania, ich budowę, możliwości i ograniczenia, związane z nimi filtry oraz narzędzia przestrzeni użytkownika. Wskazałem struktury danych jądra, na których operują dyscypliny i opisałem nagłówki protokołów IP, TCP oraz UDP. Wyjaśniłem zasadę działania algorytmu Filtra kubełka żetonów, który stał się inspiracją dla stworzonej w ramach tej pracy aplikacji.

Informacje te są niezbędne do zrozumienia projektu aplikacji przedstawionego w kolejnym rozdziale.

Rozdział 2

Projekt FireQOS

W niniejszym rozdziale opisany zostanie projekt aplikacji, nazywanej dalej FireQOS, służącej do kontroli wykorzystania pasma sieciowego przez aktywne gniazda sieciowe w Linuksie. Zaprezentowana architektura opiera się na mechanizmach dostępnych w jądrze Linuksa, omówionych w rozdziale 1. Podrozdział 2.7 zawiera opis kilku sposobów komunikacji pomiędzy przestrzeniami jądra i użytkownika, przeanalizowanych pod kątem wykorzystania w FireQOS. W przedostatnim podrozdziale przedstawiłem pomysł alternatywnego rozwiązania wraz z krótkim porównaniem do zrealizowanego projektu.

2.1. Założenia projektowe

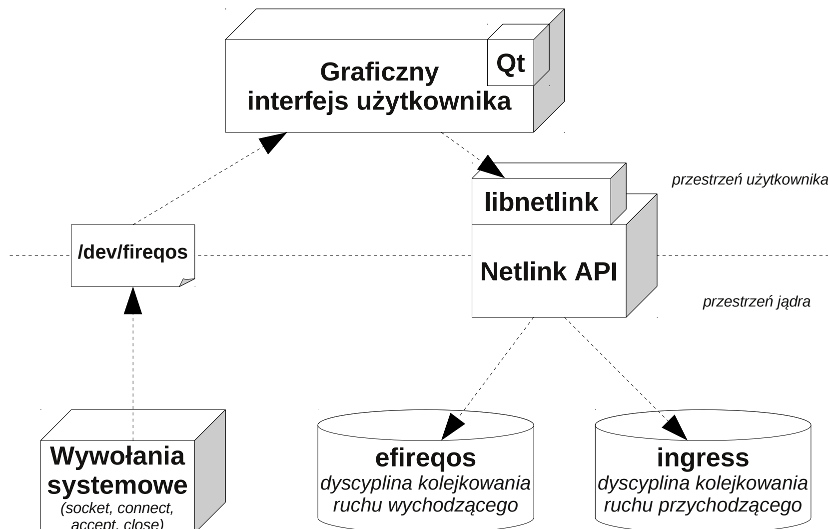
Założenia projektowe FireQOS to:

1. możliwość ograniczania pasma wykorzystywanego przez programy korzystające z sieci przy pomocy protokołów TCP, UDP, IP per połączenie,
2. zmiana ustawień w trakcie działania programu (tzn. już nawiązanych połączeń),
3. prosty graficzny interfejs użytkownika,
4. brak modyfikacji kodu jądra Linuksa.

2.2. Architektura

Po uwzględnieniu podanych założeń i ograniczeń zaprojektowałem architekturę FireQOS, składającą się z czterech głównych elementów:

- urządzenia znakowego `fireqosdev` służącego do informowania o nowych gniazdach sieciowych, nawiązywanych połączeniach oraz o zamknięciu gniazda,
- zmodyfikowanej dyscypliny kolejkowania `ingress` (patrz rozdział 1.6.4),
- dyscypliny kolejkowania dla ruchu wychodzącego – `efireqos`, opartej na pomyśle Filtra kubełka żetonów (patrz rozdział 1.6.2),
- graficznego interfejsu użytkownika (dalej nazywanego GUI, od ang. *Graphical User Interface*) służącego do prezentacji i kontroli ograniczeń pasma.



Rysunek 2.1: Architektura i komunikacja elementów FireQOS

Powiązania i kierunek komunikacji pomiędzy poszczególnymi elementami systemu są przedstawione na rysunku 2.1.

Do komunikacji pomiędzy GUI i pozostałymi modułami FireQOS wybrałem dwa rozwiązania po przeanalizowaniu dostępnych mechanizmów, opisanych w rozdziale 2.7. Informacje o zmianach w gniazdach sieciowych przesyłane są za pomocą urządzenia znakowego. Dyscypliny kolejkowania, ze względu na kwestie synchronizacji we współbieżnym środowisku jądra Linuksa, korzystają z Netlink API (patrz rozdział 2.7.4) za pośrednictwem metody `change` (patrz rozdział 1.6.5).

W kolejnych rozdziałach przedstawione są projekty elementów FireQOS.

2.3. Urządzenie znakowe `fireqosdev`

Urządzenie znakowe `fireqosdev` służy do informowania GUI o nowych gniazdach sieciowych, nawiązywanych połączeniach oraz o zamknięciu gniazda. Dzięki temu GUI może zaprezentować te informacje użytkownikowi oraz pozwolić mu określić limity dla poszczególnych gniazd i przesłać je do dyscyplin kolejkowania.

GUI mogłoby posłużyć się dostępnym w Linuksie poleceniem `netstat` lub plikami `/proc/net/tcp` i `/proc/net/udp`, jednak takie podejście ma co najmniej trzy wady – konieczność aktywnego odpytywania, duży narzut na pojedyncze sprawdzenie i przeglądanie za każdym razem całej listy gniazd. Bezpośrednią przyczyną tych wad jest przeglądanie wewnętrznych struktur jądra opisujących gniazda sieciowe. Ponieważ są one współdzielone, trzeba je blokować, a to powoduje zmniejszenie wydajności całego podsystemu sieciowego i zabiera relatywnie dużo czasu.

Przeprowadzone eksperymenty pokazały, że koszt użycia wymienionych w poprzednim akapicie rozwiązań jest nieakceptowalny. Wynik polecenia `time netstat -t -u`, przy 16 aktywnych połączeniach TCP/IP jest następujący:

```
real    0m2.440s
user    0m0.018s
sys     0m0.021s
```

Zarówno czas całkowity (zdarza się, że sięga 10 sekund), jak i czas systemowy – 0,02 s nie sprzyja aktywnemu odpytywaniu jądra. Podobne czasy uzyskuje się odczytując pliki `/proc/net/tcp` i `/proc/net/udp`.

Z tego powodu zrezygnowałem z założenia o niemodyfikowaniu istniejącego kodu jądra, co niestety istotnie zmniejsza szanse na popularyzację FireQOS.

W celu zmniejszenia narzutu wprowadzanego przez FireQOS do podsystemu sieciowego Linuksa zdecydowałem się zmodyfikować wywołania systemowe `socket`, `connect`, `accept4` oraz `close` dodając w ich treści wywołania własnych funkcji. Ponieważ wywołanie systemowe `close` służy również do zamykania zwykłych plików, a w szczególnym przypadku, gdy zamykane jest gniazdo sieciowe wywołuje ono funkcję `sock_release`[1], to ta ostatnia jest lepszym miejscem do umieszczenia własnego haka (ang. *hook*).

Komentarza wymaga oddzielne informowanie o tworzonych gniazdach i nawiązywanych połączeniach. Pierwszy powód to możliwość przesłania do GUI adresu drugiej strony połączenia (informacja ta jest niedostępna w momencie utworzenia gniazda). Drugą przyczyną jest różnica pomiędzy protokołami UDP i TCP, znajdująca odzwierciedlenie w typach gniazd; odpowiednio `SOCK_DGRAM` i `SOCK_STREAM`. Dla pierwszego typu komunikacja odbywa się bez zestawiania połączenia, zatem utworzenie gniazda jest w zasadzie jedynym miejscem, w którym można dodać hak. Dla gniazd typu `SOCK_STREAM` komunikacja odbywa się dopiero po nawiązaniu połączenia, nie ma więc potrzeby informować GUI wcześniej o utworzeniu gniazda. Co więcej, w przypadku gniazd oczekujących na połączenia (po wywołaniu systemowym `listen`), w chwili zaakceptowania połączenia (`accept`) tworzone jest nowe gniazdo, reprezentujące zestawione połączenie (nowa struktura `sock` – patrz rozdział 1.8.1).

Wobec tych ograniczeń w projekcie zakładam napisanie modułu jądra, który eksportuje funkcje:

- `fireqos_notify_new_sock(...)`,
- `fireqos_notify_new_connection(...)`,
- `fireqos_notify_close_sock(...)`.

Są to haki do umieszczenia we wspomnianych wywołaniach systemowych. Moduł przechowuje dostarczone w ten sposób informacje i udostępnia je za pomocą urządzenia znakowego. GUI odczytuje je w postaci struktury `fireqos_exchange`[2].

Zawiera ona informację o typie komunikatu (`FIREQOS_NEWSOCK`, `FIREQOS_NEWCONNECTION`, `FIREQOS_CLOSESOCK`), adres powiązanej struktury `sock`, identyfikator procesu `pid` oraz unię `data`. Unia `data` zawiera trzy struktury, typ komunikatu (`int type`) specyfikuje, której struktury użyć. Takie rozwiązanie pozwala wymieniać komunikaty o stałej długości, niezależnie od przekazywanych wiadomości, przy jednoczesnym efektywnym wykorzystaniu pamięci.

2.4. Dyscyplina kolejkowania dla ruchu przychodzącego

Zgodnie z tym, co zostało napisane o kontrolowaniu ruchu przychodzącego (patrz rozdział 1.3) oraz o dostępnej w Linuksie dyscyplinie kolejkowania dla ruchu przychodzącego `ingress` (patrz rozdział 1.6.4), postanowiłem ją zmodyfikować.

[1] `net/socket.c`

[2] `include/linux/fireqos.h`

Zmiana obejmuje zastąpienie części odpowiedzialnej za wykorzystanie filtrów do oznaczania pakietów (patrz rozdział 1.7) algorytmem opartym na Filtrze kubełka żetonów (patrz rozdział 1.6.2).

Ponieważ limity przepustowości są określane per gniazdo sieciowe, to niezbędne jest utrzymywanie struktury danych pozwalającej je efektywnie wyszukiwać. W tym celu zastosowałem samorównoważące się drzewo binarne, w którego węzłach przechowywane są ustawienia dla zadanego klucza – adresu struktury `sock`, jednoznacznie identyfikującego gniazdo.

Węzły omawianego drzewa przechowują struktury `limit_sk`, zawierające następujące pola:

- `int limit` – limit dla transferu danych (wyrażony w KiB/s),
- `int used` – suma odebranych bajtów,
- `s64 Rt` – suma zebranych żetonów (w tym wykorzystane), patrz rozdział o Filtrze kubełka żetonów (1.6.2),
- `s64 t_c` czas odebrania ostatniego pakietu (ang. *timestamp*),
- `struct sock * sk` wskaźnik do gniazda, którego dotyczy ograniczenie.

Niezmiennik podany w definicji algorytmu Filtra kubełka żetonów (patrz definicja 1.6.1, rozdział 1.6.2) można przekształcić do następującej postaci

$$used + qdisc_pkt_len(skb) \leqslant B + R_t + limit \times \Delta t$$

gdzie:

- $used$ – liczba odebranych bajtów,
- $qdisc_pkt_len(skb)$ – długość (w bajtach) odebranego pakietu,
- R_t – liczba bajtów możliwa do odebrania, wynikająca z przypływu nowych żetonów pomiędzy kolejnymi pakietami,
- Δt – różnica czasu pomiędzy bieżącym i poprzednim zaakceptowanym pakietem (w ns).

Podejmowanie decyzji o zaakceptowaniu lub odrzuceniu pakietu przebiega w następujących krokach:

1. Obliczenie liczby nowych żetonów = $limit \times \Delta t$.
2. Ewentualne zmniejszenie łącznej liczby żetonów ($R_t + limit \times \Delta$) do maksymalnej sugerowanej dla danego interfejsu sieciowego wartości. Maksimum to jest obliczane w momencie utworzenia instancji dyscypliny, jako iloczyn rozmiaru maksymalnej wielkości pakietu i sugerowanej długości kolejki dla interfejsu (są to parametry dostępne w strukturze danych opisującej urządzenie sieciowe i zależą od typu łącza i jego przepustowości).
3. Sprawdzenie podanej wyżej nierówności i przekazanie odpowiedniej wartości.

Dla niektórych pakietów przychodzących może nie istnieć gniazdo sieciowe (np. dla rutowanych, uszkodzonych lub specjalnie spreparowanych). Ruch ten jest traktowany jako jedna ogólna klasa (identyfikowana przez wskaźnik do gniazda równy `NULL`).

2.5. Dyscyplina kolejkowania dla ruchu wychodzącego

Moduł jądra Linuksa implementujący dyscyplinę kolejkowania dla ruchu wychodzącego, nazywaną dalej `efireqos`, to element FireQOS zajmujący się przydziałem pasma dla pakietów wychodzących. Podobnie jak zmodyfikowana dyscyplina `ingress`, przedstawiona w poprzednim rozdziale, `efireqos` jest oparty na algorytmie Filtra kubełka żetonów i używa podobnej wewnętrznej struktury danych.

Ponieważ dyscyplina kolejkowania dla ruchu wychodzącego pełni również funkcję kolejki dla wysyłanych pakietów, konieczne jest rozszerzenie opisanej struktury `limit_sk`, przedstawionej w projekcie modyfikacji dyscypliny `ingress`.

Pierwszą modyfikacją jest dodanie struktury `sk_buff_head`[3] – stanowi ona atrapę kolejki prostej dla pakietów powiązanych z danym gniazdem sieciowym i służy do przechowywania pakietów (struktur `sk_buff` – patrz rozdział 1.8.2) pomiędzy wywołaniami `enqueue` oraz `dequeue`.

Druga modyfikacja wiąże się z implementacją funkcji `dequeue`. Funkcja ta wybiera pakiet do wysłania przeglądając swoje wewnętrzne kolejki dla wszystkich gniazd sieciowych. Dla każdej kolejki sprawdza, czy jest dostępny pakiet i odpowiednia dla niego liczba żetonów. Brak żetonów powoduje obliczenie minimalnego czasu potrzebnego na ich uzupełnienie (patrz rozdział 1.6.2). Jeśli zostanie znaleziony pakiet, który można wysłać, to jest on przekazywany przez funkcję `dequeue` i usuwany z wewnętrznej kolejki (wspominany `sk_buff_head`), jeśli nie – sprawdzane są kolejne gniazda.

Ponieważ operacje na dyscyplinie kolejkowania są wykonywane w sekcji krytycznej (zapewnianej przez jądro Linuksa), niemożliwe jest dodanie pakietu do dyscypliny w trakcie działania `dequeue`. Przejrzenie wszystkich kolejek jest wobec tego jednoznaczne z brakiem pakietu do natychmiastowego wysłania.

Aby uniknąć głodzenia gniazd wystarczy przeglądać ich listę od miejsca następującego po ostatnim udanym pobraniu pakietu do wysyłki.

W przypadku metod dyscypliny kolejkowania bardzo ważny jest możliwie krótki czas wykonania. Zatem elementy występujące w drzewie gniazd warto połączyć listą cykliczną, dla której czas przejścia do następnika jest mniejszy od analogicznego czasu dla drzewa binarnego. Kolejność elementów na tej liście nie ma znaczenia, ważny jest ustalony porządek. Przewaga drzewa samorównoważącego się nad listą objawia się przy dodawaniu pakietu (funkcja `enqueue`) – pozwala ona w czasie logarytmicznym odnaleźć kolejkę dla gniazda. Zatem odpowiednią strukturą danych dla tej dyscypliny kolejkowania jest drzewo samorównoważące się, którego węzły są połączone w listę, co pozwala wykonać obie metody (`enqueue` i `dequeue`) efektywnie.

Kuszący może wydawać się pomysł dodania wskaźnika do struktury `limit_sk` w strukturze `sock`, dzięki czemu, zamiast wyszukiwania węzła w drzewie binarnym, można by sięgnąć pod wskazywany adres. Jednak korzystanie z takiego wskaźnika jest związane z ryzykiem sięgnięcia do zwolnionej pamięci, jeśli dojdzie do sytuacji wyścigu opisanej w projekcie graficznego interfejsu użytkownika (patrz rozdział 2.6).

2.6. Graficzny interfejs użytkownika

Projekt graficznego interfejsu użytkownika opiera się na wzorcu projektowym Model-Widok-Kontroler. Model zawiera listę wszystkich aktywnych gniazd w systemie, pogrupowanych wg procesów, które je utworzyły.

[3] include/linux/skbuff.h#L114

Widok prezentuje informacje zawarte w modelu wraz z określonymi przez użytkownika limitami i pozwala je edytować.

Za komunikację z pozostałymi elementami FireQOS odpowiadają dwie klasy. Pierwsza odczytuje informacje o zmianach w gniazdach sieciowych (z urządzenia znakowego `fireqosdev`, omówionego wcześniej). Ponieważ odczyt może wiązać się z zasypianiem w oczekiwaniu na nowe komunikaty, to musi odbywać się w oddzielnym wątku. Druga z klas służy do przygotowywania i wysyłania informacji o ustawionych limitach do dyscyplin kolejkowania za pomocą Netlink (patrz rozdział 2.7.4).

Z informowaniem GUI o utworzeniu i zamknięciu gniazd sieciowych wiąże się sytuacja wyścigu, która ma wpływ na projekt dyscyplin (wspomniany brak możliwości umieszczenia w strukturze `sock` wskaźników do struktur opisujących limity dla danego gniazda). Wynika ona z braku informacji zwrotnej od GUI po zamknięciu gniazda, tzn. może się zdarzyć, że zanim GUI odbierze informację o zamknięciu gniazda i zwolnieniu używanych przez nie struktur danych, wyśle do dyscypliny nowy limit. Z tego powodu dyscyplina nie może zaglądać pod przekazany adres struktury `sock`. Tym samym nie miałaby dostępu do struktury `limit_sk`.

2.7. Komunikacja pomiędzy przestrzeniami jądra i użytkownika

Istotną częścią projektu jest wybór metody komunikacji pomiędzy GUI i elementami FireQOS działającymi w przestrzeni jądra.

Do przesyłania limitów do dyscyplin kolejkowania zastosowałem funkcję `change` udostępnianą przez dyscypliny (patrz rozdział 1.6.5). Ubocznym efektem takiej komunikacji jest potwierdzanie odebranego komunikatu przez dyscyplinę (poprzez wartość przekazywaną przez `change`).

Na potrzeby komunikacji pomiędzy GUI i funkcjami powiadamiającymi o zmianach gniazd sieciowych rozważałem rozwiązania opisane w kolejnych podrozdziałach. Ostatecznie wybór padł na urządzenie znakowe, ze względu na najmniejszy narzut spośród wszystkich rozwiązań.

2.7.1. Urządzenia znakowe

Urządzenia znakowe to jeden z prostszych sposobów na wymianę danych pomiędzy przestrzeniami jądra i użytkownika. Pozwalają stworzyć plik urządzenia, który jest odczytywany i zapisywany przez programy dokładnie tak samo, jak zwykły plik. Bardzo podobne do urządzeń znakowych, zarówno w działaniu, jak i implementacji, jest udostępnianie plików przez `procfs` oraz `sysfs`.

Zalety:

- możliwość usypiania procesu w oczekiwaniu na dostępność zasobów,
- kontrola uprawnień dostępu do plików dla użytkowników i ich grup,
- prosta implementacja.

Wady:

- konieczność sprawdzania długości komunikatu,
- konieczność obsługi kursora pliku,
- buforowanie zapisu wymaga dodatkowej uwagi.

2.7.2. Pomocnicze wywołania przestrzeni użytkownika

Pomocnicze wywołania przestrzeni użytkownika (ang. *Usermode-helper API*) to Linuksowy mechanizm pozwalający z poziomu jądra Linuksa uruchamiać programy w przestrzeni użytkownika i przekazywać ich wynik działania do kodu wykonywanego w przestrzeni jądra. Uruchomiany program wykonywany jest z uprawnieniami administratora (ang. *root*) w dość mocno ograniczonym i odizolowanym od innych programów środowisku. Głównym przeznaczeniem jest zautomatyzowanie ładowania potrzebnych modułów jądra Linuksa, wywoływanie skryptów związanych z oszczędzaniem energii i przekazywanie zdarzeń związanych ze sprzętem. Szczegółowy opis dostępnych funkcji można znaleźć w [UsermodeHelper].

Zalety:

- możliwość prostego odbioru potwierdzenia dostarczenia komunikatu (poprzez wartość przekazaną przez wywoływany program),
- wywoływany program, może od razu wykonać konkretną akcję, np. wysłać limit do dyscypliny.

Wady:

- kosztowne przełączanie kontekstu wykonania,
- ograniczone środowisko wywoływanego programu,
- niewygodnie konstruowanie tablicy z argumentami do wywołania,
- bezpieczeństwo.

2.7.3. Kolejki komunikatów Systemu V

Kolejki komunikatów Systemu V to mechanizm komunikacji międzyprocesorowej obsługiwany przez jądro Linuksa. Pozwala on na umieszczanie komunikatów w kolejkach, które następnie inne procesy mogą pobrać. Wykorzystanie go poza kontekstem użytkownika jest bardzo utrudnione – brakuje stosownych funkcji; te dostępne wymagają jako argumentu identyfikatora przestrzeni nazw procesu.

Zalety:

- komunikaty stanowią odrębne całości,
- możliwość ustalenia priorytetów komunikatów.

Wady:

- bardzo skomplikowane użycie poza kontekstem użytkownika.

2.7.4. Netlink

Netlink to mechanizm asynchronicznej dwukierunkowej komunikacji międzyprocesowej, dostępnej również z poziomu jądra Linuksa (patrz [Netlink], RFC 3549). Korzysta z gniazd sieciowych rodziny `AF_NETLINK` i pozwala na stosunkowo prostą komunikację za pomocą wywołań systemowych `sendmsg()` i `recvmsg()`. Głównym przeznaczeniem Netlink jest konfiguracja podsystemu sieciowego (np. ustawianie adresów interfejsów sieciowych, tablicy rutowania) oraz odbieranie informacji diagnostycznych. W szczególności narzędzia z pakietu *iproute2* (patrz rozdział 1.9) korzystają z Netlink.

Zalety:

- komunikaty stanowią odrębne całości,
- możliwość wysyłania pakietów rozgłoszeniowych (do wielu odbiorców),
- dyscyplina kolejkowania może zawierać funkcję `change`, wywoływaną w momencie przyjścia komunikatu (nie trzeba implementować obsługi gniazda sieciowego Netlink).

Wady:

- zbyt duża liczba komunikatów powoduje przepadanie niektórych z nich,
- konieczność przydzielenia unikatowego numeru dla usługi, który może kolidować z innymi nieoficjalnymi rozszerzeniami.

2.8. Możliwe rozwiązania alternatywne i ulepszenia

2.8.1. Pole typu usługi w nagłówku protokołu IP

Nagłówek protokołu IP w wersji 4 zawiera pole określające ważność pakietu i preferencje nadawcy (patrz rozdział 1.2.1). Dyscypliny kolejkowania mogłyby korzystać z jego wartości. Dla ruchu wychodzącego można spróbować zastosować priorytety zależne od preferencji nadawcy (konstruktora) pakietu. Dyscyplina dla ruchu przychodzącego mogłaby ignorować limity w przypadku pakietów oznaczonych jako ważne lub tych z ustawioną flagą niezawodnego połączenia.

Znaczenie tego ulepszenia jest niewielkie wobec faktu, że wiele ruterów stosowanych w Internecie ze względów optymalizacyjnych ignoruje wartość pola typu usługi i traktuje wszystkie pakiety jednakowo.

2.8.2. Monitorowanie gniazd i połączeń

Zamiast urządzenia znakowego informującego o tworzonych i zamykanych gniazdach sieciowych oraz otwieranych połączeniach, warto rozważyć wykorzystanie Netlink (patrz rozdział 2.7.4). Wydaje się on idealnym rozwiązaniem do informowania graficznego interfejsu, działającego w przestrzeni użytkownika, o utworzeniu i zamknięciu gniazda sieciowego oraz o nawiązaniu nowego połączenia. Tym bardziej, że pozwala zrealizować komunikację rozgłoszeniową, co pozwoliłoby obok interfejsu kontrolowanego przez użytkownika uruchomić aplikację, która na podstawie własnych kryteriów mogłaby automatycznie określać limity.

Wadą wykorzystania w tym celu Netlink jest konieczność przydzielenia numeru protokołu, który mógłby kolidować z innymi nieoficjalnymi łatkami, nakładanymi na jądro Linuksa przez dystrybucje i tym samym doprowadzić w połączeniu z nimi do niepożądanych interferencji pomiędzy różnymi programami. Z drugiej strony, gdyby kod ten został włączony do głównej gałęzi rozwojowej Linuksa, problem ten przestałby istnieć.

2.8.3. Haki w wywołaniach systemowych

Alternatywą dla umieszczenia wywołań funkcji własnego modułu w wywołaniach systemowych jest umieszczenie w nich haków Netfiltra. Netfilter powstał jako odpowiedź na nieuporządkowany mechanizm haków w podsystemie sieciowym Linuksa 2.2. Celem było stworzenie jasnego i przejrzystego rozwiązania pozwalającego przechwytywać przepływające dane na kolejnych etapach ich przemieszczania się. Korzystanie z Netfiltra polega na dołączeniu swojej

funkcji do łańcucha haków, które są kolejno wywoływane w określonych miejscach funkcji podsystemu sieciowego.

Podstawowym zastosowaniem Netfiltra jest modyfikacja (ew. odrzucanie poprzez wbudowaną w jądro Linuksa ścianę ogniową) przechodzących pakietów. Wobec tego wykorzystanie go do informowania o zmianach gniazd sieciowych (na potrzeby projektowanego rozwiązania) jest nie do końca zgodne z przeznaczeniem tego mechanizmu.

Z drugiej strony wyposażenie wywołań systemowych `socket`, `accept`, `connect`, `close` w wywołania Netfiltra pozwoliłoby zrealizować wymaganą dla FireQOS funkcjonalność w postaci modułu jądra, bez konieczności dalszej ingerencji w kod Linuksa.

2.8.4. Filtr pakietów

Alternatywnym rozwiązaniem dla własnych dyscyplin kolejkowania byłoby zaimplementowanie filtra, który dopasowywałby przekazywane pakiety (struktury `sk_buff`) do zadanego gniazda sieciowego. Filtr taki można wykorzystać w Linuksowej (niezmodyfikowanej) dyscyplinie `ingress` oraz w jednej z dyscyplin dla ruchu wychodzącego z klasami, np. w Hierarchicznych kubełkach żetonów (patrz rozdział 1.6.3). Wraz z narzędziem `tc` (patrz rozdział 1.9.1) lub bezpośrednio Netlinkiem można wtedy budować konfigurację w sposób zbliżony do zaprojektowanego rozwiązania.

Największą wadą takiego pomysłu jest znacznie większy koszt obliczeniowy. Dla n aktywnych i limitowanych połączeń sieciowych, koszt znalezienia struktury opisującej ograniczenia dla danego połączenia w prezentowanym rozwiązaniu, przy zastosowaniu zrównoważonego drzewa binarnego wynosi $O(log(n))$. Gdyby zastosować mechanizm filtrów, w którym każdy z n filtrów jest kolejno przykładany do pakietu, aż jeden z nich zgłosi trafienie, koszt obliczeniowy wzrósłby do $O(n)$.

2.9. Podsumowanie

W tym rozdziale omówiłem projekt aplikacji służącej do ograniczania szybkości aktywnych gniazd sieciowych w Linuksie. Przedstawiłem dostępne mechanizmy komunikacji pomiędzy przestrzeniami jądra i użytkownika oraz zwięźle oceniłem ich wady i zalety. Zaprezentowałem pomysły alternatywnych projektów wraz z ich oceną.

Rozdział 3

Implementacja FireQOS

W niniejszym rozdziale opisana zostanie implementacja projektu aplikacji FireQOS przedstawionego w rozdziale 2. Ponadto szczegółowy opis funkcji wykorzystywanych w dyscyplinie kolejkowania ruchu przychodzącego może być użyteczny dla kogoś, kto chciałby zaimplementować inną dyscyplinę tego typu.

3.1. Urządzenie znakowe `fireqosdev`

Urządzenie znakowe `fireqosdev` ma postać modułu jądra Linuksa, którego kod znajduje się w pliku `net/sched/fireqos_dev.c`. Ze względu na haki w wywołaniach systemowych niemożliwe jest skompilowanie go w postaci dynamicznie ładowanego modułu.

Moduł eksportuje funkcje wymienione w rozdziale 2.3, za pomocą których wywołania systemowe przekazują informacje o nowych i zamykanych gniazdach sieciowych oraz o nowych połączeniach. Ponadto moduł utrzymuje dwie listy ze strukturami gotowymi do przekazania w wywołaniu funkcji `read`. Pierwsza przechowuje komunikaty typów `FIREQOS_NEWSOCK` i `FIREQOS_NEWCONNECTION`, druga – `FIREQOS_CLOSESOCK`. W takiej kolejności są przekazywane do procesu czytającego urządzenie znakowe (tzn. dopóki pierwsza lista jest niepusta, komunikaty z drugiej pozostają ukryte). Sugeruje to możliwość zagłodzenia. Komunikaty o zamknięciu gniazd można jednak pominąć. Interfejs będzie wtedy wyświetlał nieistniejące połączenia i pozwalał określać dla nich limity, co ze względu na niezależną od tego rozwiązania sytuację wyścigu opisaną w rozdziale 2.6 nie stanowi dodatkowego problemu.

Odczyt urządzenia znakowego polega na próbie opuszczenia semafora Linuksowego (którego wartość jest równa liczbie oczekujących komunikatów) i usunięciu przesłanej do użytkownika struktury `fireqos_exchange` z listy.

Niewielkim usprawnieniem mogłoby być przeglądanie i ew. usuwanie zbędnych elementów z listy z nowymi gniazdami i połączeniami w momencie zamknięcia gniazda, efektywna implementacja takiego rozwiązania wymagałaby jednak dodatkowego zrównoważonego drzewa binarnego do szybkiego wyszukiwania elementów na liście (struktury podobnej do przedstawionej w rozdziale 2.5).

3.2. Dyscyplina kolejkowania dla ruchu przychodzącego

Modyfikacja dyscypliny kolejkowania dla ruchu przychodzącego `ingress`, której projekt został przedstawiony w rozdziale 2.4, wiąże się z trzema zagadnieniami:

- implementacją pomocniczej struktury danych (samorównoważącego się drzewa binarnego),

- ustalaniem adresu gniazda sieciowego, do którego kierowany jest odebrany pakiet,
- wydajną implementacją algorytmu Filtra kubełka żetonów.

Spośród różnych typów samorównoważących się drzew binarnych wybrałem drzewa czerwono-czarne. Decyzja ta była podyktowana dostępnością implementacji tych drzew w Linuksie. Ze względu na jej ogólność niezbędne jest dopisanie własnych funkcji do dodawania, wyszukiwania i usuwania węzłów drzewa. W dokumentacji[1] są przedstawione przykłady takich funkcji wraz z wywołaniami równoważącymi drzewo.

W celu ustalenia gniazda, do którego należy struktura `sk_buff` przekazana jako argument funkcji `enqueue`, teoretycznie można posłużyć się dostępnym w niej wskaźnikiem `sk`. Eksperymenty pokazały jednak, że dla przychodzących pakietów pole `sk` jest równe `NULL` nawet wtedy, gdy istnieje gniazdo w systemie, do którego powinien trafić ten pakiet. Takie podejście wynika z optymalizacji. Ustalenie gniazda jest względnie szybką operacją dzięki zastosowaniu tablic mieszających (ang. *hash-tables*) oraz pamięci podręcznej ostatnio używanych połączeń (patrz [Grzegórski], rozdział 3.1.3), wciąż jednak warto je opóźnić na wypadek decyzji o odrzuceniu pakietu przez jeden z filtrów (w oryginalnej implementacji).

Analiza kodu funkcji `tcp_v4_rcv()`[2] prowadzi do funkcji `struct sock * __inet_lookup_skb()`[3], która wydaje się idealnie nadawać do ustalenia gniazda powiązanego z otrzymanym pakietem. Jednakże, ze względu na konieczność wyłączenia przerwań programowych na używanym procesorze (`local_bh_disable()` oraz `local_bh_enable()`, zdefiniowane w `include/asm/softirq.h`) i z powodu przyszłych zmian z kodzie Linuksa wygodniej i bezpieczniej jest użyć funkcji `inet_lookup()`[4].

To rodzi jednak kolejny problem – `inet_lookup()` przyjmuje jako parametry adresy IPv4 i numery portów zamiast wskaźnika na `struct sk_buff`. Można je wprawdzie uzyskać za pomocą funkcji `skb_pull()`[5], po upewnieniu się, że długość pakietu jest wystarczająca – `pskb_may_pull()`[6], ale spowoduje to usunięcie pobieranych nagłówków protokołów IP i TCP z bufora, co uniemożliwi późniejsze dostarczenie pakietu. Rozwiązaniem jest skorzystanie z funkcji `struct iphdr *ip_hdr()` oraz `struct tcphdr *tcp_hdr()` z własnoręcznie wyliczonym przesunięciem (ang. *offset*). Nagłówek protokołu TCP jest bezpośrednio za nagłówkiem protokołu IP.

Konieczne jest również sprawdzenie analogicznie do funkcji `ip_rcv()`[7], czy odebrany pakiet jest poprawny, tzn.

1. długość jest równa co najmniej długości nagłówka IP,
2. wersja protokołu (`iphdr.version`) jest równa 4 (dla IPv4),
3. zgadza się suma kontrolna,
4. długość pakietu określona w nagłówku odpowiada rzeczywistej długości.

Realizuje to kod w pliku `net/ipv4/ip_input.c`, w liniach 414 – 430. Funkcja służąca do odbierania pakietów IPv6 – `ip6_rcv()`[8] sprawdza jedynie wersję protokołu.

[1] `Documentation/rbtree.txt`
[2] `net/ipv4/tcp_ipv4.c#L353`
[3] `include/net/inet_hashtables.h#L377`
[4] `include/net/inet_hashtables.h#L362`
[5] `net/core/skbuff.c#L1052`
[6] `include/linux/skbuff.h#L1149`
[7] `net/ipv4/ip_input.c#L379`
[8] `net/ipv6/ip6_input.c#L58`

Analogicznie treść funkcji `udp_rcv()`[9] prowadzi do wywołania funkcji `udp4_lib_lookup()`[10] pozwalającej ustalić gniazdo, do którego trafią dane przesyłane przy użyciu protokołu UDP.

Należy podkreślić, że choć funkcje `inet_lookup()` i `udp4_lib_lookup()` nie ustawiają wskaźnika `sk` struktury `sk_buff` w celu wykorzystania w przyszłości, to w zasadzie brak ku temu przeciwwskazań. Spowodowałoby to zmniejszenie narzutu obliczeniowego wprowadzanego przez dyscyplinę kolejkowania dla ruchu przychodzącego do absolutnego minimum.

Do wywołania funkcji `inet_lookup()` niezbędne jest też przekazanie wskaźnika do przestrzeni nazw odpowiadającej sieci (`struct net`) oraz indeksu urządzenia sieciowego (ponieważ połączenia są przechowywane per interfejs). Struktura `struct sk_buff` zawiera wprawdzie wskaźnik do urządzenia sieciowego, jednak eksperymenty pokazują, że nie można na nim polegać przy implementacji dyscypliny dla ruchu przychodzącego. Można jednak skorzystać z własności drugiego argumentu przekazywanego do funkcji `enqueue()` – struktura związana z dyscypliną kolejkowania również posiada wskaźnik do powiązanego urządzenia sieciowego i dzięki funkcjom `qdisc_dev()` oraz `dev_net()` zdobycie wymaganych parametrów dla `inet_lookup()` jest wykonalne.

Z algorytmem Filtra kubełka żetonów zaadaptowanym na potrzeby modyfikowanej dyscypliny wiąże się konieczność zamiany jednostek. Czas jest wyrażony w nanosekundach i reprezentowany liczbą 64-bitową (funkcja `ktime_to_ns()`), limit dla transferu jest 32-bitową wartością wyrażoną w KiB/s. W celu uniknięcia dzielenia przez $\frac{1024}{10^9}$ przy zmianie jednostek, stosuję przesunięcie bitowe o 20 w prawo. Pomysł ten jest zaczerpnięty z implementacji dyscyplin kolejkowania dostępnych w jądrze Linuksa. Można obliczyć, że błąd takiego przesunięcia (dzielenia przez 2^{20}) jest niewielki oraz jest łatwy do skompensowania w GUI (jednorazowo), zaś przy wykonywaniu `enqueue` unika się kosztownego dzielenia liczb 64-bitowych (dla każdego pakietu).

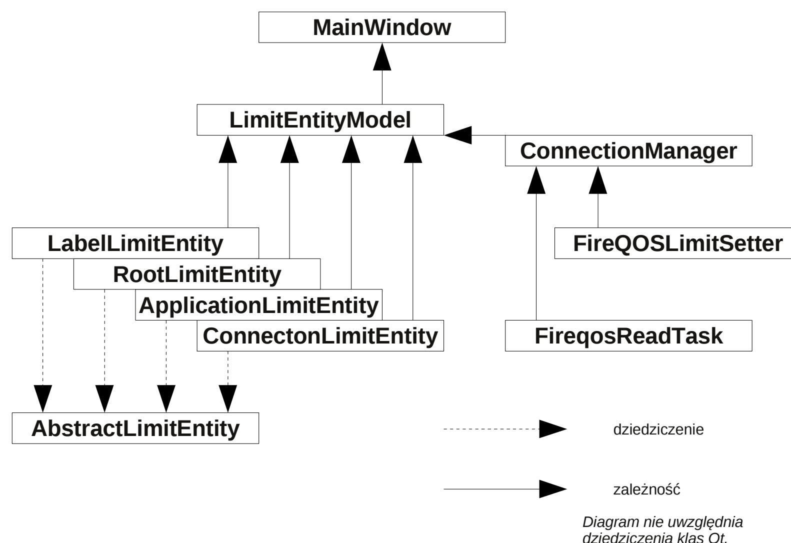
Przy implementacji dyscypliny kolejkowania konieczne jest utrzymywanie statystyk wskazanych w rozdziale 1.6.5. Dla ruchu przychodzącego, jeśli pakiet jest odrzucany, należy zwiększyć wartość pola `qstats.drop` struktury `Qdisc` (reprezentującej instancję dyscypliny) przekazanej jako argument do `enqueue`.

3.3. Dyscyplina kolejkowania dla ruchu wychodzącego

Dyscyplina kolejkowania dla ruchu wychodzącego pełni funkcję kolejki pakietów. Jej implementacja wiąże się z określeniem pojemności – przyjęcie zbyt wielu pakietów do dyscypliny uniemożliwi ich niezwłoczne dostarczenie. Wartość pojemności ściśle zależy od rodzaju interfejsu sieciowego i jego szybkości. Można ją odczytać w strukturze opisującej urządzenie sieciowe, do którego podłączona jest instancja dyscypliny (`qdisc_dev(sch)`). Jej atrybut `tx_queue_len` określa maksymalną dopuszczalną liczbę pakietów oczekujących w dyscyplinie.

Istnieją dwie miary zapełnienia kolejek – liczba pakietów oraz suma długości wszystkich pakietów. Ponieważ projekt (patrz rozdział 2.5) zakłada utrzymywanie wielu kolejek (dla każdego gniazda), to wygodniejsza jest pierwsza miara. Dzięki niej można sformułować warunek przyjęcia pakietu do dyscypliny: liczba pakietów w kolejce dla gniazda sieciowego, z którego pakiet został wysłany, nie może przekroczyć łącznej dopuszczalnej liczby pakietów dzielonej przez liczbę kolejek. Jednakże przynajmniej jeden pakiet dla danego gniazda musi zostać zaakceptowany. W przeciwnym przypadku gniazda o niewielkich limitach mogłyby zapełnić dyscyplinę, uniemożliwiając dostęp do urządzenia sieciowego pakietom, które mogłyby zostać wysłane szybciej. Pakiety przekraczające limit pojemności kolejki są odrzucane, podobnie jak

[9] `net/ipv4/udp.c#L1611`
[10] `net/ipv4/udp.c#L514`



Rysunek 3.1: Diagram klas graficznego interfejsu użytkownika FireQOS

w przypadku dyscypliny dla ruchu przychodzącego. W ten sposób została zaimplementowana funkcja `enqueue`.

Właściwy algorytm planowania i kształtowania jest zawarty w funkcji `dequeue`. Obliczanie liczby dostępnych żetonów przebiega analogicznie, jak w dyscyplinie kolejkowania ruchu przychodzącego (patrz rozdział 3.2). Modyfikacja polega na obliczaniu czasu potrzebnego na zebranie żetonów dla każdej kolejki, w której znajduje się przynajmniej jeden pakiet, ale liczba żetonów jest niewystarczająca, aby go wysłać. Minimum z tych czasów określa minimalny czas, jaki `dequeue` musi odczekać, by móc cokolwiek przekazać jako swój wynik. W tym celu uruchamia mechanizm budzenia z obliczoną zwłoką i przekazuje wartość `NULL` jako wynik. Ponieważ dostępna funkcja `qdisc_watchdog_schedule`[11] posługuję się milisekundami do określania czasu (które za pomocą makra i przesunięcia bitowego przekształca na nanosekundy), stworzyłem jej odpowiednik `qdisc_watchdog_schedule_ns`. Dzięki temu nie tracę precyzji obliczonego czasu.

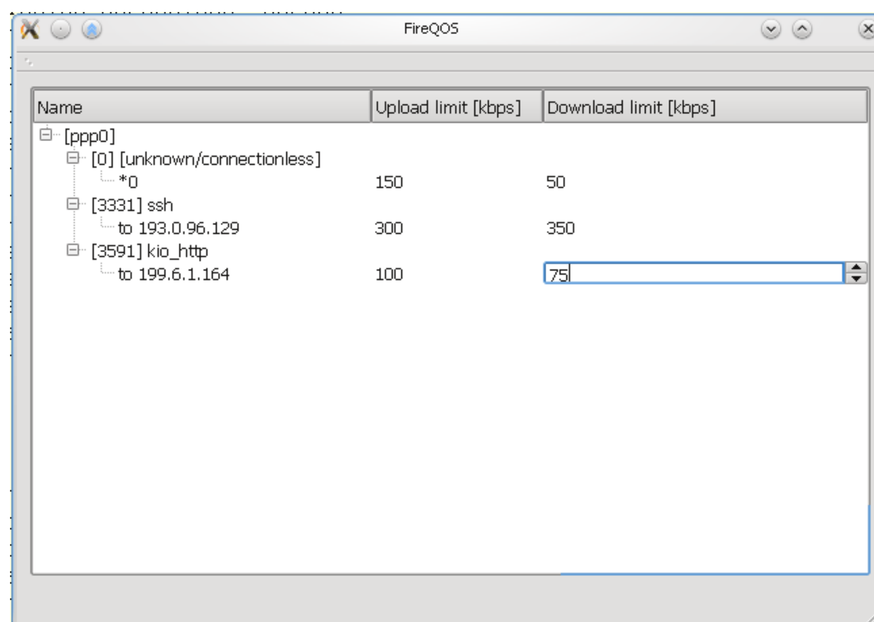
Dla pakietów rutowanych przez system niemożliwe jest wskazanie gniazda sieciowego. Ruch ten jest identyfikowany przez wskaźnik gniazda równy `NULL`.

3.4. Graficzny interfejs użytkownika

Implementacja graficznego interfejsu użytkownika powstała w języku programowania C++ przy użyciu biblioteki *Qt*[12] w wersji 4.6.1. Wybór *Qt* był podyktowany łatwością implementacji wzorca projektowego Model-Widok-Kontroler oraz wykorzystania mechanizmu asynchronicznych zadań wykonywanych w tle i rozbudowanego mechanizmu sygnałów i gniazd (ang. *signals and slots*). Diagram klas (z pominięciem dziedziczenia po klasach *Qt*) został przedstawiony na rysunku 3.1

Graficzny interfejs użytkownika został zaprezentowany na rysunku 3.2.

[11] net/sched/sch_api.c#L474
[12] `http://qt.nokia.com`



Rysunek 3.2: Graficzny interfejs użytkownika FireQOS

3.4.1. Model-Widok-Kontroler

Modelem w GUI jest drzewo, na którego kolejnych poziomach są: interfejs sieciowy, programy, gniazda sieciowe. Reprezentują je następujące klasy: `RootLimitEntity`, `ApplicationLimitEntity` oraz `ConnectionLimtEntity`. Wszystkie dziedziczą po klasie abstrakcyjnej `AbstractLimitEntity`, co ułatwia implementację metod, które musi obsługiwać klasa `LimitEntityModel`. Ta ostatnia klasa jest modelem wg kryteriów zrębu (ang. *framework*) Model-Widok-Kontroler *Qt*. Zarządza ona obiektami wymienionych wcześniej klas i udostępnia ich dane (nazwa, ustawione limity) widokowi.

Do prezentacji modelu wykorzystałem widok drzewa dostępny w *Qt* – `QTreeView`.

3.4.2. Zadania wykonywane w tle

Ponieważ odczytywanie komunikatów z urządzenia znakowego `fireqosdev` wiąże się z oczekiwaniem na semaforze, aby program mógł działać, musi odbywać się to w oddzielnym wątku. W tym celu wystarczy stworzyć klasę dziedziczącą po `QRunnable` i zaimplementować w niej wirtualną metodę `void run()`. W jej treści można umieścić pętlę, która odczytuje kolejne komunikaty i emitując sygnał (patrz rozdział 3.4.3) powoduje uwzględnienie zmian w modelu.

Aby uruchomić wątek wystarczy stworzyć instancję i przekazać ją jako argument do wywołania `QThreadPool::globalInstance()->start()`. Powrót z metody `run` jest równoznaczny z zakończeniem wykonywania wątku i jego usunięciem.

W FireQOS klasą realizującą odczyt urządzenia znakowego `fireqosdev` jest `FireqosReadTask`

3.4.3. Sygnały i gniazda

Sygnały i gniazda to mechanizm komunikacji międzyprocesowej dostępny w *Qt*.

Sygnał ma postać deklaracji funkcji w klasie (bez implementacji). Obiekt emituje go za

pomocą makra `emit`, np. `emit hasChanged();`. Gniazdo to z kolei zwykła funkcja wykonywana jako odpowiedź na odebrany sygnał.

Deklaracje sygnałów i gniazd wyglądają tak, jak w przykładowym kodzie:

```
class LimitEntityModel : public QAbstractItemModel
{
Q_OBJECT

  /* ... */

signals:
  void hasChanged();

public slots:
  void closeSocket(void *sock, int pid);

  /* ... */

};
```

Do połączenia sygnału z odpowiednim gniazdem służy funkcja `QObject::connect`. Aby sygnał i gniazdo dały się połączyć muszą przekazywać wynik takiego samego typu oraz mieć identyczne argumenty – te same typy danych w tej samej kolejności (nazwy są pomijane). Sygnał wysłany przez konkretny obiekt (przekazany jako argument do `connect`) spowoduje wywołanie funkcji-gniazda w obiekcie docelowym. Jeśli sygnał i gniazdo są typu innego niż `void`, to wartość przekazana przez gniazdo zostanie odesłana do obiektu, który wyemitował sygnał. Dzięki temu korzystanie z sygnałów i slotów niewiele się różni od zwykłego wywoływania funkcji. Mechanizm sygnałów i gniazd nie jest ograniczony do pojedynczego procesu; podłączanie sygnałów i gniazd poprzez usługę D-Bus[13] jest tylko nieznacznie bardziej skomplikowane.

W graficznym interfejsie FireQOS sygnały i gniazda są wykorzystywane do przekazywania informacji odczytanych z urządzenia znakowego `fireqosdev` (patrz rozdział 2.3). W ten sposób działający asynchronicznie wątek `FireqosReadTask`, który w pętli odczytuje kolejne komunikaty, informuje model `LimitEntityModel` o pojawieniu się nowych gniazd sieciowych i połączeń oraz o zamknięciu gniazd.

3.5. Przesyłanie limitów do dyscyplin kolejkowania

Dyscypliny kolejkowania otrzymują komunikaty o zmianach limitów dla poszczególnych gniazd w postaci struktur `fireqos_limit_exchange`[14]. Są one przesyłane za pomocą Netlink (patrz rozdział 2.7.4). Dyscypliny nie zawierają jednak bezpośredniej obsługi gniazd Netlink, a jedynie otrzymują umieszczone w nich dane za pomocą funkcji `change`.

[13]D-Bus to usługa mająca uprościć komunikację pomiędzy procesami i stworzyć jeden standard dla takiej komunikacji, `http://www.freedesktop.org/wiki/Software/dbus`

[14]`include/linux/fireqos.h`

Po przeanalizowaniu kodu narzędzia `tc` (patrz rozdział 1.9.1) wyodrębniłem wywołania potrzebne do przesłania komunikatów o zmianach limitów prędkości za pomocą biblioteki `libnetlink`, wchodzącej w skład pakietu *iproute2*.

Przekazanie struktury `fireqos_limit_exchange` do dyscypliny przebiega w następujących krokach:

1. otworzenie biblioteki *libnetlink* (odpowiednik utworzenia gniazda sieciowego),
2. przygotowanie struktury opisującej żądanie `nlmsghdr`; w tym wskazanie interfejsu sieciowego i dyscypliny,
3. przygotowanie struktury `fireqos_limit_exchange`,
4. zbudowanie z wymienionych struktur żądania,
5. wywołanie funkcji `rtnl_talk()`,
6. zamknięcie biblioteki *libnetlink*.

Wartość `0` przekazana przez funkcję `rtnl_talk()` świadczy o pomyślnym dostarczeniu komunikatu do dyscypliny kolejkowania.

3.6. Sposób użycia FireQOS

Aby korzystać z FireQOS dla interfejsu sieciowego eth0 należy:

1. uruchomić jądro Linuksa z łatką obsługującą urządzenie znakowe `fireqosdev`,
2. utworzyć plik `/dev/fireqos` poleceniem:
   `mknod /dev/fireqos c $(grep fireqosdev /proc/devices | cut -f 1 -d " ") 0`
   i nadać mu uprawnienia do odczytu przez administratora,
3. uruchomić GUI poleceniem: `FireQOS eth0`

Wykonanie przedstawionych operacji (zwłaszcza pierwszej) wymaga dobrej znajomości systemu operacyjnego Linux, co przeczy kierowaniu FireQOS do domowych użytkowników. Jednakże te operacje mogą zostać ukryte przez autora dystrybucji Linuksa. Podczas nakładania innych łatek na jądro Linuksa dodanie jeszcze jednej od FireQOS jest proste. Tworzenie pliku `/dev/fireqos` i uruchomienie graficznego interfejsu można dodać do plików startowych systemu.

Instalacja FireQOS na już działającym systemie będzie kłopotliwa dla docelowych użytkowników programu, ale jeśli sięgną oni po odpowiednio przygotowaną dystrybucję Linuksa – nie będą musieli się z nią samodzielnie zmagać.

3.7. Podsumowanie

W tym rozdziale opisałem szczegóły implementacji dyscyplin kolejkowania zaprojektowanych w rozdziale 2. Przedstawiłem krótko wykorzystane mechanizmy z biblioteki *Qt*, na której oparty jest graficzny interfejs użytkownika. Podałem również sposób uruchomienia FireQOS.

Rozdział 4

Testy

Rozdział ten zawiera opis i wyniki testów opracowanego programu.

4.1. Opis środowiska testowego

Testy FireQOS przeprowadziłem na dwóch komputerach połączonych siecią Ethernet o przepustowości 100Mbit. Pierwszy komputer, na którym testowałem FireQOS jest wyposażony w dwurdzeniowy procesor AMD Turion taktowany zegarem 1,9 GHz oraz 2 GiB pamięci RAM. Drugi komputer, który pełnił rolę nadawcy lub odbiorcy opisanego dalej programu `ttcp` jest wyposażony w procesor Intel Pentium III taktowany zegarem 1 GHz i 384 MiB pamięci RAM.

Do testowania ograniczania i przepustowości użyłem programu `ttcp`, uznawanego za standardowe narzędzie do pomiaru szybkości łącza. Korzysta on z protokołu TCP i próbuje przesłać możliwie wiele danych od klienta do serwera.

Na potrzeby testów dodałem do GUI opcję trybu testowego. Polega ona na automatycznym ustawianiu limitów (przekazywanych jako argumenty w wierszu poleceń) dla wszystkich nowych gniazd i połączeń.

Jednym z testów było sprawdzenie możliwości zmiany ograniczenia pasma w trakcie przesyłania danych przez dane połączenie. Do tego celu wykorzystałem program `scp`, za pomocą którego przesyłałem duży plik pomiędzy komputerami.

Każdy test był powtarzany pięć razy.

Podczas testów FireQOS obie dyscypliny były podłączone do interfejsu sieciowego, z tym, że ta która nie była testowana miała limity ustawiane powyżej maksymalnej przepustowości karty sieciowej.

4.2. Wyniki testów

Przeprowadziłem trzy rodzaje testów: sprawdziłem zdolność dyscyplin do ograniczania pasma, ich przepustowość oraz skuteczność zmiany ograniczenia w czasie.

4.2.1. Testy ograniczania pasma

Testy ograniczania pasma polegały na ustawieniu limitu dla wszystkich gniazd na kolejno 50 KiB/s, 500 KiB/s i 5000 KiB/s. W tym samym czasie dyscyplina dla ruchu w przeciwnym kierunku miała ustawiony limit na 100000 KiB/s.

Limit: 50 KiB/s

ingress	**próba**	efireqos
39.09 KiB/s	1.	46.40 KiB/s
38.78 KiB/s	2.	46.73 KiB/s
38.67 KiB/s	3.	46.40 KiB/s
38.87 KiB/s	4.	46.48 KiB/s
38.91 KiB/s	5.	46.43 KiB/s
38.86 KiB/s	**średnia**	46.48 KiB/s
0.10 KiB/s	**odchylenie standardowe**	0.10 KiB/s

Rysunek 4.1: Wynik dla limitu 50 KiB/s

Limit: 500 KiB/s

ingress	**próba**	efireqos
446.22 KiB/s	1.	462.14 KiB/s
445.15 KiB/s	2.	462.38 KiB/s
444.77 KiB/s	3.	462.11 KiB/s
444.32 KiB/s	4.	462.52 KiB/s
444.85 KiB/s	5.	462.13 KiB/s
445.06 KiB/s	**średnia**	462.25 KiB/s
0.63 KiB/s	**odchylenie standardowe**	0.14 KiB/s

Rysunek 4.2: Wynik dla limitu 500 KiB/s

Tabelki 4.1, 4.2 i 4.3 zawierają wyniki pomiarów dla obu dyscyplin kolejkowania (z niezależnych eksperymentów, zestawione razem celem łatwiejszego porównania).

4.2.2. Testy przepustowości dyscypliny

Drugim rodzajem testu było ustawienie wszystkich limitów w dyscyplinach kolejkowania na 100000 KiB/s (ok. 8 razy szybciej od teoretycznej szybkości interfejsu).

Postanowiłem porównać osiągnięte szybkości z parą dyscyplin, `pfifo` (kolejka prosta) oraz niezmodyfikowanej dyscypliny `ingress` z Linuksa.

Wyniki zestawione są w tabelach 4.4 i 4.5.

Limit: 5000 KiB/s

ingress	**próba**	efireqos
4493.32 KiB/s	1.	4630.91 KiB/s
4492.84 KiB/s	2.	4631.81 KiB/s
4493.29 KiB/s	3.	4631.91 KiB/s
4493.41 KiB/s	4.	4631.66 KiB/s
4478.88 KiB/s	5.	4631.72 KiB/s
4490.34 KiB/s	**średnia**	4631.60 KiB/s
5.73 KiB/s	**odchylenie standardowe**	0.34 KiB/s

Rysunek 4.3: Wynik dla limitu 5000 KiB/s

Limit: 100000 KiB/s (powyżej możliwości interfejsu sieciowego)		
`ingress`	**próba**	`ingress` **(FireQOS)**
10093.89 KiB/s	1.	9594.25 KiB/s
10440.11 KiB/s	2.	9916.52 KiB/s
10394.76 KiB/s	3.	9329.68 KiB/s
10433.82 KiB/s	4.	9235.33 KiB/s
10437.95 KiB/s	5.	9357.73 KiB/s
10360.10 KiB/s	**średnia**	9486.70 KiB/s
134.13 KiB/s	**odchylenie standardowe**	245.27 KiB/s

Rysunek 4.4: Porównanie dyscyplin dla ruchu przychodzącego

Limit: 100000 KiB/s (powyżej możliwości interfejsu sieciowego)		
`pfifo`	**próba**	`efireqos`
10977.47 KiB/s	1.	10513.95 KiB/s
10517.78 KiB/s	2.	10291.03 KiB/s
10909.76 KiB/s	3.	10541.96 KiB/s
9436.51 KiB/s	4.	10434.67 KiB/s
10451.68 KiB/s	5.	10032.55 KiB/s
10458.64 KiB/s	**średnia**	10362.83 KiB/s
551.54 KiB/s	**odchylenie standardowe**	186.72 KiB/s

Rysunek 4.5: Porównanie dyscyplin dla ruchu wychodzącego

4.3. Testy zmiany ograniczenia pasma w czasie

Jednym z ważniejszych założeń FireQOS jest możliwość zmiany wartości ograniczenia w trakcie przesyłania danych przez dane połączenie. Do przetestowania tej funkcjonalności posłużyło mi polecenie `scp` służące do kopiowania plików przy użyciu protokołu ssh. Test polegał na zmianie ograniczenia kolejno na 50 KiB/s, 500 KiB/s, 2500 KiB/s, 500 KiB/s. Po ustawieniu kolejnej wartości i ustabilizowaniu się wyświetlanej chwilowej szybkości pobierania lub wysyłania, przejście do nowej linii pozwala zachować statystyki na konsoli. Wyniki testu są przedstawione na rysunkach 4.6 i 4.7.

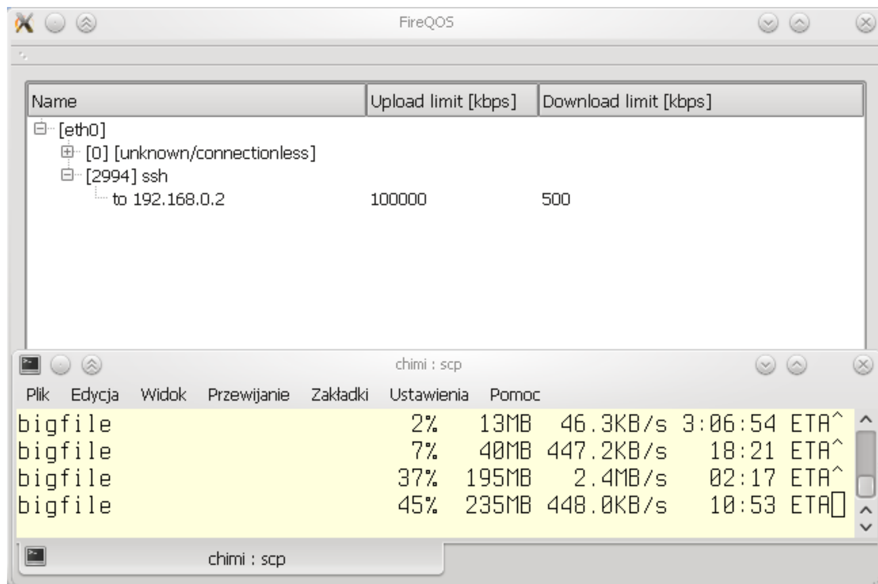
4.4. Analiza i interpretacja wyników

Uwagę zwraca powtarzalność wyników ograniczania pasma oraz skalowalność zastosowanego algorytmu, szczególnie dobrze widoczna w dyscyplinie kolejkowania ruchu wychodzącego.

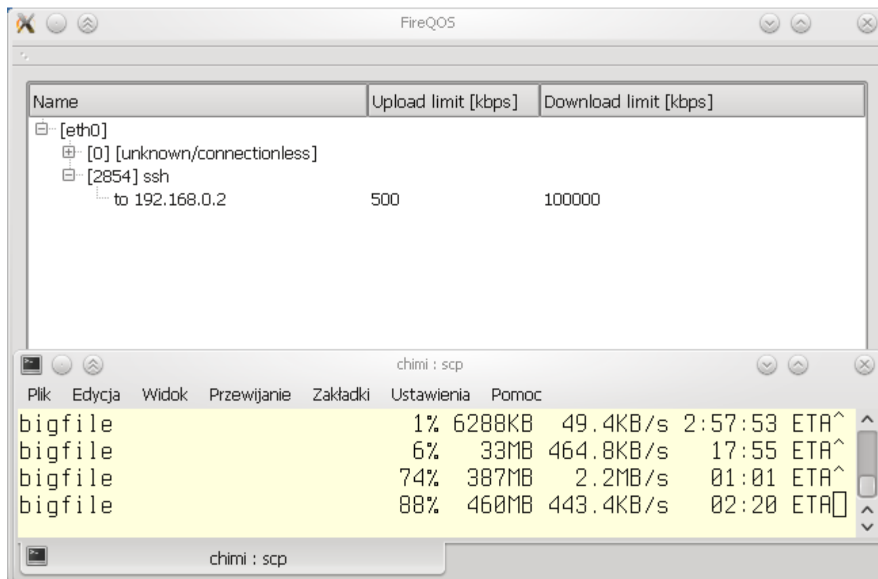
Komentarza wymaga zaniżenie wartości ograniczenia w stosunku do ustawionego limitu. Na czas prób usunąłem z GUI niwelowanie błędu wynikającego z przesunięcia bitowego zastosowanego w celu optymalizacji (patrz rozdział 3.2).

Mniejsze przepustowości osiągane przez dyscyplinę dla ruchu przychodzącego wynikają prawdopodobnie z dość losowego odrzucania przychodzących pakietów i konieczności ich retransmisji przez protokół TCP. Uzasadnia to pogląd, że kontrolowanie ruchu przychodzącego to działanie po niewłaściwej stronie łącza (patrz rozdział 1.3).

Średnia przepustowość zaimplementowanych przeze mnie dyscyplin jest nieco gorsza, niż wybranych do porównania dyscyplin z jądra Linuksa. Wytypowałem je ponieważ nie zawie-



Rysunek 4.6: Testy zmiany ograniczenia pasma w czasie dla ruchu przychodzącego



Rysunek 4.7: Testy zmiany ograniczenia pasma w czasie dla ruchu wychodzącego

rają żadnych algorytmów i trudno jest im dorównać w kwestii czasu wykonywania operacji. Należy zwrócić uwagę, że oryginalna dyscyplina `ingress` podczas testów nie wywoływała żadnych filtrów, a ta należąca do FireQOS ustalała dla każdego pakietu danych adres gniazda i przeliczała żetony. Wobec tego faktu uśredniony wynik gorszy o ponad 0,8 MiB/s uznaję za akceptowalny.

W przypadku dyscypliny dla ruchu wychodzącego okazuje się, że zaprojektowana struktura danych wnosi bardzo nieznaczne opóźnienia i wypada porównywalnie do prostej kolejki.

Test zmiany ograniczenia pasma w trakcie przesyłania danych wypadł pomyślnie. Zmiana szybkości jest widoczna niemal natychmiast po ustawieniu nowego limitu szybkości. Zarówno zwiększanie, jak i zmniejszanie wartości ograniczenia nie sprawia żadnych problemów.

Rozdział 5

Podsumowanie

Postawiony we wstępie problem stworzenia programu pozwalającego w prosty sposób kontrolować wykorzystanie łącza sieciowego przez poszczególne połączenia (gniazda) uważam za rozwiązany. Stworzona przeze mnie aplikacja składa się z dyscyplin kolejkowania ruchu przychodzącego i wychodzącego, graficznego interfejsu użytkownika oraz modułu jądra Linuksa, który powiadamia o nowych gniazdach sieciowych, połączeniach oraz o zamknięciu gniazd. Algorytm zastosowany w dyscyplinach kolejkowania jest oparty na pomyśle Filtra kubełka żetonów (dyscypliny, której działanie przeanalizowałem i opisałem w pracy).

Niestety nie udało się uniknąć modyfikacji jądra Linuksa ze względu na wydajność rozwiązania, przez co szanse na spopularyzowanie FireQOS zmalały. Żałuję tym bardziej, że tego typu programów w zasadzie nie ma (poza wspomnianymi we wstępie *cFosSpeed* i *NetLimiterem*, dostępnymi tylko dla MS Windows), a mógłby to być jeden z argumentów w dyskusji, jaki system operacyjny najlepiej odpowiada potrzebom zwykłego użytkownika.

Wydaje się, że wskazane w pracy alternatywne sposoby zmodyfikowania jądra Linuksa, szczególnie połączenie gniazd Netlink oraz haków Netfiltra, przy odpowiednio dużym zainteresowaniu ze strony społeczności tego systemu operacyjnego, miałyby stosunkowo duże szanse na włączenie do głównej gałęzi rozwojowej Linuksa.

Testy pokazały, że stworzone dyscypliny kolejkowania są dokładne, efekty ich działania są powtarzalne, a wprowadzany przez nie narzut obliczeniowy jest akceptowalny dzięki odpowiednio zaprojektowanym wewnętrznym strukturom danych. Dyscyplina kolejkowania dla ruchu wychodzącego dorównuje przepustowością kolejce prostej, co jest dużym osiągnięciem wobec faktu, że wykonuje znacznie więcej wewnętrznych obliczeń. Część z nich została zoptymalizowana – zamiast kosztownego dzielenia liczb 64-bitowych stosuję znacznie szybsze przesunięcie bitowe.

Z myślą o użytkownikach domowych, przechodzących na Linuksa z systemu MS Windows, do których jest adresowany stworzony program, można opracować interfejs dla wbudowanej w Linuksa ściany ogniowej. Taka funkcjonalność poniekąd jest dostępna w FireQOS – poprzez ustawienie limitu równego 0, jednak wyskakujące okienka przy próbie nawiązania połączenia przez program, który nie został wcześniej oznaczony jako zaufany, bardziej przypominałyby rozwiązania dla konkurencyjnego systemu.

Poza możliwymi ulepszeniami, opisanymi w projekcie aplikacji, warto również rozważyć dopracowanie obsługi protokołu IP w wersji 6.

Większą część wysiłku związanego ze stworzeniem tego programu pochłonęło eksperymentowanie z podsystemem sieciowym i analiza jego funkcji. Zrozumienie tego podsystemu jest dość trudnym zadaniem z powodu niepełnej i najczęściej nieaktualnej dokumentacji. Wymusza to konieczność analizowania kodu jądra Linuksa, w którym nazwy funkcji nie zawsze

są oczywiste, a efekty uboczne ich wywołania bywają zaskakujące.

Mam nadzieję, że sporządzony przeze mnie opis dyscyplin kolejkowania ruchu sieciowego i powiązanych z nimi zagadnień przynajmniej przez jakiś czas będzie na tyle aktualny i kompletny, by umożliwić komuś stworzenie własnej dyscypliny kolejkowania.

Stworzony program mam zamiar opublikować na listach dyskusyjnych poświęconych Linuksowi i aktywnie go rozwijać.

Dodatek A

Zawartość płyty dołączonej do pracy

Na płycie znajdują się:

- w katalogu `praca` – elektroniczna wersja pracy magisterskiej,
- w katalogu `Linux` – kod źródłowy systemu operacyjnego Linux 2.6.33.4,
- w katalogu `Qt` – kod źródłowy biblioteki Qt wraz z dokumentacją i środowiskiem programistycznym,
- w katalogu `FireQOS-kernel` – łatka na jądro Linuksa z dyscyplinami kolejkowania oraz urządzeniem znakowym `fireqosdev`,
- w katalogu `FireQOS-GUI` – kod źródłowy graficznego interfejsu użytkowa FireQOS,
- w katalogu `testy` – wyniki przeprowadzonych testów wraz z opisem ich uzyskania,

Bibliografia

[Understanding] Christian Benvenuti, *Understanding Linux Network Internals*, O'Reilly, December 2005

[Grzegórski] Dariusz Grzegórski, *Mechanizmy komunikacji sieciowej w wybranych systemach operacyjnych*, Marzec 2003

[Netlink] Kevin K. He, *Kernel Korner - Why and How to Use Netlink Socket*, `http://www.linuxjournal.com/article/7356`

[TCQoS] Jennifer Hou, *CS 498 Lecture 9: Traffic Control for QoS*, `http://www.cs.uiuc.edu/class/fa05/cs498hou/lectures/lecture10.pdf`

[LARTC] Bert Hubert, *Linux Advanced Routing & Traffic Control HOWTO*, `http://tldp.org/HOWTO/Adv-Routing-HOWTO/`

[UsermodeHelper] Tim Jones, *Invoking user-space applications from the kernel*, `http://www.ibm.com/developerworks/linux/library/l-user-space-apps/index.html`

[LinuxKernel05] Robert Love, *Linux Kernel Development Second Edition*, Sams Publishing, January 12, 2005

[RCU] Paul E. McKenney, *Read-Copy-Update*, `http://www.rdrop.com/users/paulmck/RCU/`

[Networks] Larry L. Peterson, Bruce S. Davie, *Computer networks: a systems approach*, Elsevier, 2003

[Sieci] Andrew S. Tanenbaum, *Sieci komputerowe*, Helion, Październik 2004

[Linux2.6.33.4] Linus B. Torvalds i inni, *Kod źródłowy Linuksa 2.6.33.4*, `http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.33.4.tar.bz2`

[LNNDAPI] *Linux Networking and Network Devices APIs* [w:] *Linux Kernel HTML Documentation (DocBook)* `http://ww2.cs.fsu.edu/~rosentha/linux/2.6.26.5/docs/DocBook/networking/index.html`