802.11 (Tytuł do ustalenia)

Marcin Harasimczuk

22 stycznia 2012

Spis treści

0.1 Cel pracy

Celem niniejszej pracy inżynierskiej jest specyfikacja wymagań i implementacja narzędzia pomiarowego *hop-sniffer* umożliwiającego obserwacje wybranych zjawisk zachodzących podczas komunikacji systemów w standardzie 802.11. Główny nacisk kładziony jest na pomiar zależności czasowych między zdarzeniami charakteryzującymi dany scenariusz komunikacyjny. Poprzez zdarzenie rozumiem fakt nadania lub odebrania ramki 802.11. Ramki są podstawowym elementem protokołu komunikacyjnego, więc możliwość obrazowania zależności czasowych między nimi daje szansę ustalenia trwania dowolnych zjawisk charakteryzujących 802.11.

Skupienie na pomiarze parametrów czasowych wynika z przyświecającego pracy celu analizy zastosowania komunikacji bezprzewodowej w wymianie danych systemów czasu rzeczywistego. Systemy takie w swoim obrębie pracują w sposób deterministyczny i zgodny z przyjętymi założeniami. Problem pojawia się w sytuacji, gdy nadchodzi potrzeba ustanowienia łącza komunikacyjnego między nimi. W celu zachowania założeń opóźnienia wprowadzane przez takie łącze muszą spełniać te same rygorystyczne wymagania co same systemy. Możliwość pomiarów oferowana przez program *hop-sniffer* jest pomocna nie tyle w odpowiedzi na pytanie, czy komunikacja w standardzie 802.11 jest deterministyczna, lecz jakie mogą być przybliżone czasy trwania wybranych zjawisk w ustalonych warunkach medium.

Z pośród wybranych scenariuszy w komunikacji bezprzewodowej praca ta skupia się głównie na roamingu 802.11 stacji klienckiej i będącym jego integralną częścią zjawisku przełączania kanału radiowego. Zjawisko to jest ważne głownie z perspektywy stacji mobilnych, których interfejsy radiowe, w celu zachowania łączności, są zmuszone do zmian częstotliwości pracy zgodnie z kanałem działania punktu dostępowego obsługującego aktualnie odwiedzany obszar. Sformułowanie sposobu pomiaru czasu trwania roamingu stacji klienckiej posłuży mi do wyznaczenia wymagań stawianych aplikacji *hop-sniffer*. Wymagania te są podstawą do wyboru technik programistycznych, używanych bibliotek i środowiska działania programu.

Ostatecznie niniejsza praca wymaga wykorzystania aplikacji *hop-sniffer* w realizacji uprzednio sformułowanej procedury pomiarowej w celu wyciągnięcia wniosków co do czasu przełączania kanału radiowego podczas roamingu stacji klienckiej w standardzie 802.11. Wnioski dotyczą głównie wpływu zastosowanych ustawień i wybranego środowiska pod kątem możliwego zastosowania w systemach czasu rzeczywistego.

0.2 Wprowadzenie do dziedziny

0.2.1 Problemy systemów czasu rzeczywistego

Badanie komunikacji systemów czasu rzeczywistego wymaga analizy dostępnych rozwiązań na poziomie oprogramowania. Jest to czynność niezbędna ze względu na różnice w implementacji i stosowane techniki programistyczne. W swojej pracy skupiam się na rozwiązaniach *open-source* i systemie *Linux* ze względu na dostępność, zgodność ze standardem POSIX oraz otwarty kod źródłowy.

Systemy z rodziny *open-source* charakteryzują się możliwością stopniowania wsparcia dla procesów czasu rzeczywistego ([8]). Zaczynając od podstawowej dystrybucji systemu *Linux*, poprzez różnorodne opcje konfiguracyjne jądra w wersji 2.6 i kończąc na koncepcji współdzielenia zasobów sprzętowych.

Niniejszy rozdział poświęcam na wprowadzenie do problemów systemów czasu rzeczywistego oraz przegląd odpowiadających im rozwiązań. Ze względu na tematykę pracy koncentruję się również na kwestii komunikacji sieciowej (implementacji stosu IP).

System operacyjny Linux (wersja jądra 2.6).

Za analizą zastosowania systemu Linux jako systemu czasu rzeczywistego przemawia jego szeroka dostępność i niski koszt zastosowania. Aplikacje pracujące w reżimie czasu rzeczywistego mogą być pisane zgodnie ze standardem POSIX co wyklucza dodatkowy narzut związany z przyswajaniem nowych interfejsów programistycznych.

Jądro w wersji 2.4, ze względu na zastosowanie BKL (ang. Big Kernel Lock) wymuszało sekwencyjne wykonanie procesów działających w jego kontekście. BKL jest globalnym *spin-lock'iem* zajmowanym przez proces, który zaczyna wykonywać kod jądra (np. w wyniku wywołania systemowego) i zwalnianym po powrocie do przestrzeni użytkownika. Takie podejście zapewnia, że w kontekście jądra może wykonywać się tylko jeden wątek. Sekwencyjność całkowicie wyklucza możliwość zastosowania jako system czasu rzeczywistego.

Wersja 2.6 znacząco poprawia ten stan rzeczy. Dzięki lokalnemu blokowaniu zasobów wątek jądra może zostać wywłaszczony tylko w ściśle określonych miejscach. Zmniejszanie opóźnień odbywa się poprzez systematyczne zastępowanie *spin-lock'ów* blokadami typu *mutex*. *Mutex* pozwala na lepsze wykorzystanie czasu procesora, gdyż wątek oczekujący na wejście do sekcji krytycznej nie wykonuje aktywnego oczekiwania. Mechanizm *spin-lock* jest lepszym wyborem w sytuacji kiedy jest pewne, że narzut związany z przełączaniem kontekstu jest większy niż przewidywany czas aktywnego oczekiwania. *Spin-lock* jest zatem dobrym rozwiązaniem jeśli mamy do czynienia z ograniczoną współbieżnością, w przeciwnym przypadku powoduje

duże straty czasu procesora w skutek aktywnego oczekiwania.

Podobnie jak w wersji 2.4 istnieje ryzyko długich opóźnień, jeśli zadanie o niskim priorytecie zablokuje obsługę przerwań.

Jądro w wersji 2.6 wprowadza nowy algorytm szeregowania procesów nazwany po prostu *O(1)*. Algorytm ten zapewnia, że czas szeregowania nie zależy od ilości procesów. Nie zmienia to faktu, że proces nie będący procesem czasu rzeczywistego może z powodzeniem zablokować możliwość wywłaszczenia blokując obsługę przerwań.

Poważniejsze zmiany w jądrze systemu Linux dostępne są po wybraniu odpowiednich opcji kompilacji. Każda kolejna opcja zwiększa granulację blokad w kodzie jądra co przekłada się na zwiększenie liczby punktów, w których może ono zostać wywłaszczone. Zmiany te, w połączeniu z jawnym oznaczeniem przez programistę zadań pracujących w reżimie czasu rzeczywistego, mają wpływ na parametry opisujące działanie planisty. Pogorszeniu może ulec przepustowość (ang. throughput), rozumiana jako ilość procesów, które kończą swoją pracę na jednostkę czasu. Parametr ten ulega pogorszeniu, gdyż np. polityka SCHED_FIFO nie obsługuje podziału czasu (ang. time-slicing). Z drugiej strony można oczekiwać poprawy wydajność (ang. scheduler efficiency) rozumianej jako parametr odwrotnie proporcjonalny do opóźnień wprowadzanych przez planistę. Wydajność może wzrosnąć, gdyż SCHED_FIFO pozwala na ustalenie stałych priorytetów co w niektórych przypadkach znacząco przyspiesza harmonogramowanie. Obecnie dostępne są następujące opcje konfiguracyjne, przy czym ostatnia z nich wymaga zastosowania łatki:

- CONFIG_PREEMPT_VOLUNTARY
- CONFIG_PREEMPT
- CONFIG_PREEMPT_RT

Opcja PREEMPT_RT wzbogaca jądro o następujące możliwości:

- Możliwość wywłaszczenia w sekcjach krytycznych
- Możliwość wywłaszczenia kodu obsługi przerwań
- Wywłaszczalne obszary *blokowania przerwań*
- Dziedziczenie priorytetów dla semaforów i *spin-locków* wewnątrz jądra
- Opóźnione operacje
- Techniki redukcji opóźnień

Po zastosowaniu łatki większość kodu obsługi przerwań wykonuje się w kontekście procesu. Wyjątkiem są przerwania związane z zegarem CPU

(np. *sheduler_tick()*). Zastosowane zmiany powodują, że zadanie wykonujące *spin_lock()* może zrzec się czasu procesora, a co za tym idzie nie powinno działać przy zablokowanych przerwaniach (pojawia się zagrożenie blokadą). Jako rozwiązanie problemu przyjęto opóźnianie operacji, które nie mogą być wykonane przy zablokowanych przerwaniach do czasu ich odblokowania. Dodatkowe techniki redukcji opóźnień polegają przykładowo na rezygnacji z używania niektórych instrukcji MMX związanych z architekturą x86 (wyselekcjonowano instrukcje uznane za zbyt długie).

Techniki programistyczne w standardzie POSIX

W środowisku Linux tworzenie aplikacji czasu rzeczywistego polega na przemyślanym przeciwdziałaniu najczęstszym przyczynom długich opóźnień. Jako podstawowe przyczyny opóźnień wyróżniam:

- Brak stron kodu, danych i stosu związanych z aplikacją w pamięci (ang. *page fault*)
- Opóźnienia powstałe w skutek optymalizacji wprowadzanej przez kompilator (np. *copy-on-write*)
- Dodatkowy czas potrzebny na tworzenie nowych wątków, z których korzysta aplikacja

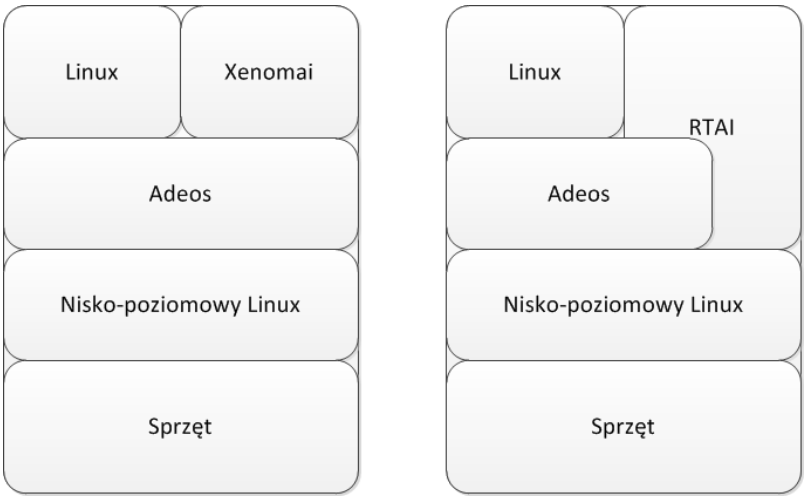
Podstawowym krokiem ku zapewnieniu, że kod aplikacji uzyska czas procesora tak szybko jak to potrzebne jest jawne oznaczenie danego procesu. Oznaczenie odbywa się poprzez wybranie trybu kolejkowania i priorytetu dla zadania. W celu oznaczenia wykorzystuję funkcję *sched_setscheduler()*, która pozwala na wybór polityki SCHED_FIFO, lub SCHED_RR. Procesy, które mają nadany stały priorytet za pomocą wywołania *sched_setscheduler()* wywłaszczają inne korzystające z metod SCHED_OTHER, SCHED_BATCH, oraz SCHED_IDLE. Wywłaszczenie procesu czasu rzeczywistego odbywa się poprzez jawne wywołanie *sched_yield()*, próbę dostępu do I/O lub przez inny proces o wyższym stałym priorytecie. SCHED_FIFO jest prostą metodą kolejkowania bez podziału czasu, zaś SCHED_RR dodatkowo przydziela procesom czasu rzeczywistego kwanty czasu.

Kolejnym krokiem jest utrzymanie w pamięci wszystkich stron kodu, danych i stosu związanych z daną aplikacją. W tym przypadku korzystam z funkcji *mlockall()*. Aby zapewnić odpowiednią ilość miejsca na stosie, oraz ustrzec się opóźnień związanych z optymalizacją kompilatora (ang. *copy-on-write*) tworzę funkcję, która alokuje zmienną automatyczną typu tablicowego. Dodatkowo, pisanie do tej zmiennej zapewnia, że cała pamięć dla niej przeznaczona będzie udostępniona przez kompilator na początku działania aplikacji.

Należy również pamiętać o tworzeniu wszelkich wątków potrzebnych do działania aplikacji na samym początku jej działania.

Xenomai i RTAI

Zarówno Xenomai, jak i RTAI są rozwiązaniami opartymi o ideę współdzielenia zasobów sprzętowych. Współdzielenie odbywa się poprzez warstwę abstrakcji sprzętowej (w tym przypadku jest to nanokernel Adeos). Adeos nie jest jednak wyłącznie niskopoziomową częścią jądra, lecz pozwala na jednoczesne uruchomienie wielu jąder, które za jego pośrednictwem współdzielą zasoby sprzętowe.



Rysunek 1: Architektura Xenomai i RTAI

Propagacja przerwań odbywa się za pośrednictwem kolejki. Kolejka jest łańcuchem (ang. *pipeline*) systemów operacyjnych, które są kolejno budzone w reakcji na otrzymane przerwanie. W przypadku Xenomai jest on umieszczony na początku kolejki i obsługuje przerwania związane z zadaniami czasu rzeczywistego. RTAI, zgodnie ze swoją polityką maksymalnej redukcji opóźnień, samodzielnie przyjmuje przerwania, a kolejki Adeos używa jedynie do dalszej propagacji nieobsłużonych przerwań (1).

Warto wspomnieć również o udostępnianej w środowisku Xenomai opcji *skórek RTOS* (ang. *real-time operating system skins*). Pozwalają one na wybór API, z którego będą korzystać uruchamiane w Xenomai aplikacje. Do wyboru są przykładowo skórki VxWorks, co ukazuje tendencje rozwoju w stronę przenośności rozwiązania.

Porównanie Linux 2.6, Xenomai, RTAI i VxWorks

Z dostępnego w publikacji [1] zestawienia systemów wynika głównie fakt, że w prostym scenariuszu, kiedy znana jest liczba (w tym przypadku wyłącznie jedna) i rodzaj pracujących aplikacji czasu rzeczywistego system Linux

w wersji 2.6 spisuje się zadowalająco w roli systemu o łagodnych ograniczeniach czasowych. W przypadku jądra systemu Linux 2.6 razem z liczbą i stopniem skomplikowania uruchamianych aplikacji rośnie również ilość kodu do przeanalizowania, w celu zapewnienia całkowitego determinizmu operacji, co szybko staje się niepraktyczne.

Interesujący jest dla mnie fakt, że gdy w rolę wchodzi dodatkowo komunikacja sieciowa, systemy *open-source* (RTAI i Xenomai) spisują się znacznie lepiej od VxWorks. Mniejsze opóźnienia są spowodowane wykorzystaniem modułu RTnet, który przebudowuje standardowy stos IP systemu linux pod kątem deterministycznej pracy.

0.2.2 Standardy 802.11 w systemach czasu rzeczywistego

Ze względu na swój charakter komunikacja bezprzewodowa jest nieprzewidywalna. Nie jesteśmy w stanie z góry założyć, ze sygnał nie zostanie zakłócony i informacja dotrze do celu. Pocieszający jest fakt, że na przestrzeni lat standard 802.11 podlegał wielu modyfikacjom i poprawkom (np. 802.11e, 802.11n). Część z tych aktualizacji dedykowana była możliwości zastosowania medium bezprzewodowego do komunikacji systemów czasu rzeczywistego. Koncentrują się one na potrzebie zapewnienia takiemu systemowi okien dostępu bezkolizyjnego i nadawaniu priorytetów w ruchu sieciowym. Zabiegi te pozwalają na osadzenie komunikacji systemów czasu rzeczywistego w zakłóconym paśmie transmisyjnym oraz ich koegzystencję z innymi stacjami.

Niniejszy rozdział poświęcony jest przeglądowi dostępnych rozwinięć standardu 802.11 pod kątem użyteczności w komunikacji systemów czasu rzeczywistego.

Problemy w 802.11 MAC - opcja DCF

Podstawowa wersja protokołu dostępu do medium (ang. *Distributed Coordination Function*, w skrócie DCF) nie uwzględnia możliwości ustalenia priorytetów. Brak priorytetów na poziomie MAC w oczywisty sposób utrudnia realizację przewidywalnej komunikacji w systemie czasu rzeczywistego. System musiałby konkurować ze wszystkimi innymi stacjami (nawet tymi, które nie są świadome jego istnienia, a więc wprowadzają wyłącznie zakłócenia).

Dodatkowo, stacje, które przy próbie dostępu napotkały zajęty kanał transmisji, muszą odczekać pewien losowy okres czasu (ang. *Backoff*). Długość okresu oczekiwania obliczana jest jako iloczyn losowej wartości z zakresu od zera do ustalonej długości CW (ang. *Contention Window*) i czasu podróży ramki w łączu (ang. *Slot Time*). Oczywiste jest, że wprowadzenie losowego parametru kłóci się z ideą deterministycznej pracy.

802.11 MAC - opcja PCF

Część powyżej przedstawionych problemów jest adresowana przez wprowadzenie protokołu dostępu PCF (ang. *Point Coordination Function*). Opcja ta wyróżnia jedną stację (typowo jest to punkt dostępu - AP), która pełni rolę koordynatora komunikacji. Koordynator uzyskuje dostęp do łącza częściej, gdyż jego czas oczekiwania między kolejnymi ramkami jest krótszy. Po uzyskaniu dostępu do łącza koordynator wybiera stację, która może rozpocząć transmisje w oknie bezkolizyjnym.

W tym przypadku problemem jest fakt, że nadająca stacja może wysłać ramkę arbitralnej długości. Punkt dostępowy rozsyła z okresem TBTT (ang. *Target Beacon Transmission Time*) ramki typu *Beacon* porządkujące transmisję danych. Przykładowo w 802.11e ramki *Beacon* zawierają ustawienia parametrów (np. TXOP) i informują inne stacje o zakończeniu okresu dostępu bezkolizyjnego. Protokół dostępu PCF nie posiada ograniczeń, które mogłyby powstrzymać stację przed pogwałceniem okresu TBTT. Jest możliwe, że stacja, która została wybrana przez AP w okresie dostępu bezkolizyjnego rozpocznie przesyłanie zbyt dużych ramek, których czas przesyłania przekroczy czas okresu TBTT. Przekroczenie czasu TBTT powoduje, że AP opóźni propagację ramek *Beacon*. Brak możliwości deterministycznego określenia długości trwania okresu dostępu bezkolizyjnego, czy też samego czasu transmisji pojedynczej stacji w tym okresie powoduje, że obsługa komunikacji systemów czasu rzeczywistego za pomocą protokołu PCF jest utrudniona.

Wsparcie dla QoS w standardzie 802.11e

Standard 802.11e wprowadził nowy protokół dostępu EDCA (ang. *Enhanced Distributed Channel Access*). Protokół ten udostępnia 4 klasy priorytetów dla ruchu. System priorytetów zbudowany jest na bazie parametrów odziedziczonych po protokole DCF.

Każda klasa dostępu posiada własny odstęp międzyramkowy AIFS (ang. *Arbitration Interframe Space*).

Czas *Backoff* nadal wyliczany jest w sposób losowy, ale w tym przypadku jest to wartość z przedziału od CWmin do CWmax (CWmin i CWmax są parametrami ustalanymi indywidualnie dla każdej klasy dostępu).

Może dojść do sytuacji, kiedy natężenie ruchu w sieci komunikacyjnej jest na tyle duże, że priorytety nie wystarczają dla zapewnienia odpowiedniej jakości połączenia dla systemu czasu rzeczywistego. W takiej sytuacji możliwe jest użycie parametru TXOP (ang. *Transmission Oportunity*).

Parametr TXOP pozwala stacji na transmisję serii ramek (ang. *burst*). Po uzyskaniu dostępu do łącza węzeł może przesyłać kolejne ramki z odstępem SIFS (ang. *short interframe space*) między porcją danych, a ramką potwierdzenia ACK. TXOP zapewnia bezkolizyjny dostęp do medium jednocześnie ograniczając maksymalny czas okresu dostępu bezkolizyjnego dla

danej stacji. Jeśli przesyłanie ramki trwałoby dłużej niż czas TXOP to ramka ta zostanie podzielona, aby zachować jakość usług.

Podsumowując, istnieją 4 parametry podlegające niezależnej regulacji w ramach każdej z klas dostępu:

- CWmin - minimalna długość losowego składnika czasu *Backoff*.
- CWmax - maksymalna długość losowego składnika czasu *Backoff*.
- AIFS - Odstęp międzyramkowy.
- Max TXOP - Maksymalny czas dostępu bezkolizyjnego po uzyskaniu medium.

CWmin i CWmax są wykorzystywane w mechaniźmie *Backoff* w celu wylosowania czasu trwania okresu wycofania stacji po kolizji. Ostateczna długość odstępu międzyramkowego wyliczana jest poprzez sumowanie czasu SIFS z iloczynem AIFS i czasu *Slot Time* łącza. Warto zauważyć, że zmniejszanie czasów oczekiwania rywalizującej stacji umożliwia jej częstszy dostęp do medium, lecz powoduje również wzrost kolizji na łączu.

Zastosowanie 802.11e w przykładowym środowisku komunikacyjnym

Standard 802.11e udostępnia następujące klasy dostępu:

- AC_VO - klasa dostępu głosowego (najwyższy priorytet)
- AC_VI - klasa dostępu wideo
- AC_BE - klasa ruchu uprzywilejowanego
- AC_BK - ruch w tle (najniższy priorytet)

W tym przypadku system czasu rzeczywistego mógłby korzystać z klasy AC_VO ([4]). Inne stacje świadome istnienia systemu mogą komunikować się w klasie AC_BK. Jeśli chodzi o stacje nieświadome istnienia systemu to można dla nich przeznaczyć klasę AC_BE.

Rozwiązania na poziomie oprogramowania Linux

Mechanim QoS (ang. *Quality Of Service*) jest realizowany w jądrze 2.6 w postaci struktury komponentów, z których każdy realizuje pewien podzbiór funkcjonalności związanych z sterowaniem ruchem sieciowym ([6]). Główne składniki modułu QoS w systemie Linux to:

- qdisc - odpowiada za politykę kolejkowania ramek na urządzeniu.

- class - umożliwia podział pakietów na klasy priorytetów w ramach qdisc.
- filter - jest elementem odpowiadającym za podział na klasy lub np. upuszczanie ramek.

Wstępnie dostępne są dwa elementy qdisc - ingress i root (egress). Główna funkcjonalności skupia się w qdisc'u root, gdyż odpowiada on za kolejkowanie ramek wychodzących (qdisc ingress umożliwia jedynie np. proste upuszczanie ramek).

Istnieją dwa typy komponentów qdisc - korzystający z klas (ang. *classfull qdisc*) i nie korzystający z klas (ang. *classless qdisc*).

Qdisc nie wykorzystujący klas pozwala na realizację prostej polityki kolejkowania typu FIFO (ang. *First-in-first-out*). Standardowo, system Linux do kolejkowania swoich ramek wykorzystuje qdisc typu *pfifo_fast*, która składa się z trzech kolejek FIFO opróżnianych wedle swojego priorytetu. *Classless qdisc* pozwala na realizację prostej polityki ograniczania częstotliwości ramek. TBF (ang. *Token Bucket Filter*) bazuje na buforze wypełnianym małymi porcjami danych (żetonami), które są konsumowane przez wysyłane ramki.

Qdisc korzystający z klas pozwala na implementację bardzo skomplikowanych struktur drzewiastych (ang. *Hierachical Token Bucket*). Każda klasa może zawierać inne klasy, przy czym w liściach drzewa następuje kształtowanie ruchu (znajdują się tam elementy qdisc). Klasy służą jedynie do odpowiedniego podziału dostępnych żetonów. Zaimplementowano również mechanizm pożyczania. Jeśli klasa dziecko wyczerpała dostępne żetony, to pożycza je od klasy rodzica.

Stos IP RTnet

RTnet jest nowym stosem IP przeznaczonym dla systemów Xenomai i RTAI. Zastępuje on standardowy stos systemu Linux i wprowadza zmiany istotne z punktu widzenia komunikacji systemów czasu rzeczywistego.

Bufor pakietu *sk_buff* został zastąpiony przez strukturę *rtskb*, która ma następujące własności:

- Stały rozmiar (zawsze maksymalny)
- Pula buforów jest alokowana na początku działania systemu dla każdej warstwy stosu
- Zawsze wraca do nadawcy, chyba, że odbiorca może zwrócić wskazanie na inny bufor z własnej puli

Inną ważną cechą RTnet jest fakt, że implementacja UDP/IP odbywa się poprzez statyczne przypisanie adresów (nie korzysta z protokołu ARP). Dla

pakietów IP przesyłanych we fragmentach potrzebne bufory *rtskb* pobierane są z puli globalnej.

0.3 Pomiar czasu przełączania kanału radiowego

Ciężko uniknąć sytuacji, w której systemy wykorzystujące do komunikacji standard *802.11* napotykają potrzebę zmiany częstotliwości (przełączenia kanału) pracy swoich interfejsów kart radiowych NIC (ang. *Network Interface Card*). Główną przyczyną podziału pasma jest wielodostęp, a więc unikanie wzajemnego zakłócania się urządzeń. Należy wziąć pod uwagę, że medium transmisyjne w środowisku przemysłowym jest zwykle wyjątkowo zaszumione w paśmie *2.4 GHz*. Dla uzmysłowienia stopnia zakłóceń wystarczy wymienić część urządzeń pracujących w paśmie *ISM* (ang. *Industrial, scientific and medical*) takich jak:

- Elektroniczne nianie
- Urządzenia Bluetooth
- Kuchenki mikrofalowe
- Alarmy samochodowe

Łatwo zauważyć jak bardzo zróżnicowane urządzenia mogą doprowadzić do niespodziewanych problemów w bezprzewodowej komunikacji systemów czasu rzeczywistego.

Warto wspomnieć, że istnieje już specyfikacja standardu pracującego w paśmie *5 GHz* ([3]), lecz nie jest on jeszcze powszechnie wspierany. Biorąc za przykład rozwiązania *open-source* można zauważyć, że standard *802.11n* jest obsługiwany przez nowe sterowniki (*ath9k* dla urządzeń firmy *Atheros*). Problemem jest natomiast fakt, że tego typu sterowniki dostępne są jedynie w najnowszych dystrybucjach systemów operacyjnych przeznaczonych dla urządzeń wbudowanych (przykładowo *OpenWrt Backfire 10.03*), które nie zawsze od początku wspierają zadowalającą gamę urządzeń. Dla przykładu nadal istnieją problemy z dostępnością tego typu sterowników dla popularnej płytki *MagicBox*.

Biorąc pod uwagę fakt zaszumienia medium transmisyjnego wnioskuję, że możliwość zmiany częstotliwości pracy interfejsu *NIC* w poszukiwaniu dogodnego kanału komunikacji jest jedną z jego kluczowych i wymagających uwagi cech. W ostatnich latach powstało wiele publikacji dotyczących możliwości adaptacji struktury sieci bezprzewodowych do panującej jakości medium komunikacyjnego ([9]). Prace te koncentrują się głównie na algorytmach dynamicznej modyfikacji częstotliwości pracy interfejsów w sieciach kratowych *WMN* (ang. *Wireless Mesh Network*). Oczywiście u podstaw zastosowanych rozwiązań leży zjawisko przełączania kanału radiowego.

Powyższe czynniki sugerują, że całkowite wyeliminowanie potrzeby przełączania kanału (zmiany częstotliwości pracy) interfejsów radiowych nie jest aktualnie osiągalne. Co więcej, udostępnianie nowych pasm częstotliwości, w sytuacji ciągle rosnącego zapotrzebowania, jest jedynie tymczasowym rozwiązaniem.

0.3.1 Przełączanie kanału radiowego

Opóźnienie związane ze zmianą częstotliwości pracy jest ważnym parametrem, gdyż w tym czasie stacja zaprzestaje reakcji na kierowane do niej dane. Ramki skierowane do stacji są tracone co w oczywisty sposób może wpłynąć na ograniczenia czasowe, w których działają komunikujące się systemy. Typowe scenariusze, w których może zajść potrzeba zmiany częstotliwości pracy interfejsu NIC to:

- Stacja kliencka w trybie *Managed* dokonuje *Roamingu* między dwoma punktami dostępowymi AP (ang. *Access Point*)
- Stacja kliencka w trybie *Managed* skanuje medium w poszukiwaniu punktów dostępowych AP (ang. *Access Point*)
- Stacja kliencka w trybie *Ad-hoc* skanuje medium po podniesieniu interfejsu lub samym przełączeniu kanału

Identyfikacja powyższych sytuacji to pierwszy krok ku specyfikacji konkretnych scenariuszy pomiarowych.

Najczęstszą przyczyną przełączania kanału jest procedura skanowania medium komunikacyjnego. Podczas skanowania stacja wysyła ramki typu *Probe Request* na każdej z dostępnych w specyfikacji ([2]) częstotliwości pracy i oczekuje na ramki *Probe Response* od punktów dostępowych, lub stacji w trybie *Ad-hoc* (w zależności od typu interfejsu NIC, czyli rodzaju docelowej sieci).

Przełączanie kanału następuje również, kiedy stacja kliencka oddala się zbyt daleko od punktu dostępowego i musi rozpocząć poszukiwanie nowego w swoim zasięgu. Jest to sytuacja zwana roamingiem i wymaga uwagi podczas rozważania systemów, w których skład wchodzą mobilne stacje, czy agenci. Obszar działania systemu może być na tyle różnorodny pod względem zakłóceń, że konieczne będzie przełączanie kanału między kolejnymi punktami dostępowymi pracującymi na różnych częstotliwościach.

0.3.2 Metodyka pomiaru

Z punktu widzenia zjawiska komunikacji w standardzie 802.11 za kluczową uznałem możliwość prowadzenia pomiarów z minimalną ingerencją w strukturę i działanie stacji. Osiągnięcie tego celu wymaga uruchomienia dodatkowej maszyny, która prowadzi nasłuch w medium komunikacyjnym.

Jedną z zalet tego typu rozwiązania jest fakt, że programistyczne środowisko pomiarowe przygotowuję tylko na jednej stacji. Jest to niezwykle ważne w przypadku, gdy w danym scenariuszu pomiarowym biorą udział systemy wbudowane (np. pełniące funkcję routerów) z ograniczonymi możliwościami instalacji rozbudowanych aplikacji i bibliotek programistycznych. Opis stosowanych metodyk pomiarowych rozpocznę od definicji podstawowych pojęć opisujących środowisko i uczestników scenariuszy. Najważniejsze pojęcia to:

- **Stacja pomiarowa**: Komputer działający pod kontrolą interakcyjnego systemu operacyjnego, na którym uruchomiona jest aplikacja nasłuchująca ruch sieciowy (ang. *sniffer*).
- **Stacja kliencka**: Komputer pełniący rolę klienta w sieci o strukturze wykorzystującej punkty dostępowe (ang. *Infrastructure mode*). Może być to zarówno komputer pod kontrolą systemu interakcyjnego, lub wbudowanego.
- **Punkt dostępowy**: Komputer pełniący w trybie infrastruktury (ang. *Infrastructure mode*) rolę stacji AP (ang. *Access Point*). Może być to zarówno komputer pod kontrolą systemu interakcyjnego, lub wbudowanego.
- **Rozwiązanie asocjacji**: Zdarzenie wysłania ramki rozwiązującej asocjację między stacją kliencką, a punktem dostępowym (ang. *Disassociation frame*).
- **Skanowanie**: Wysyłanie przez stację ramek typu *Probe Request* na wszystkich dostępnych w specyfikacji ([2]) częstotliwościach pracy.
- **Scenariusz pomiaru**: Jeden ze scenariuszy możliwych do zaistnienia podczas komunikacji stacji w standardzie 802.11, w którego czasie następuje przełączenie kanału interfejsu NIC.

0.3.3 Scenariusz pomiaru: Roaming 802.11

Roaming 802.11 to zjawisko zachodzące w sieciach, w trybie infrastruktury (ang. *Infrastructure mode*). Podstawowym zadaniem procedury jest umożliwienie stacji klienckiej odłączenia się od punktu dostępowego i podjęcia próby odnalezienia i podłączenia się do stacji o mocniejszym sygnale. W warunkach rzeczywistych sytuacja taka najczęściej jest wynikiem ruchu mobilnej stacji klienckiej (np. przemieszczającego się pracownika biura, lub agenta w systemie przemysłowym), która dociera do granicy zasięgu dotychczas używanego punktu dostępowego. Aby zachować połączenie z systemem, lub usługami (np. dostęp do internetu) maszyna musi odnaleźć inną stację pracującą w trybie AP o mocniejszym sygnale. Na procedurę roamingu 802.11 składają się następujące kroki:

– Stacja kliencka wykrywa, że poziom sygnału RF (ang. *Radio Frequency*) punktu dostępowego #1 jest poniżej progu roamingu.

– Stacja kliencka rozpoczyna nadawanie ramek rozwiązujących asocjację do punktu dostępowego #1 do momentu potwierdzenia odebrania.

– Punkt dostępowy #1 otrzymuje ramkę rozwiązującą asocjację *Disassociation frame* i usuwa stację kliencką z tablicy asocjacji.

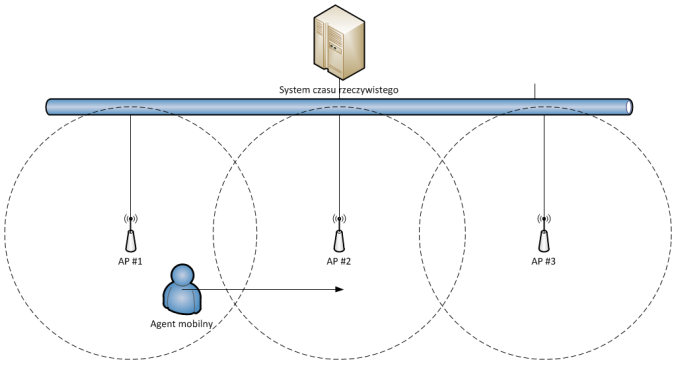
– Stacja kliencka rozpoczyna skanowanie medium komunikacyjnego i oczekuje na ramki typu *Probe Response*.

– Punkt dostępowy #2 wysyła do stacji klienckiej ramkę typu *Probe Response*

– Stacja kliencka rozpoczyna wysyłanie do punktu dostępowego #2 ramek typu *Association request*.

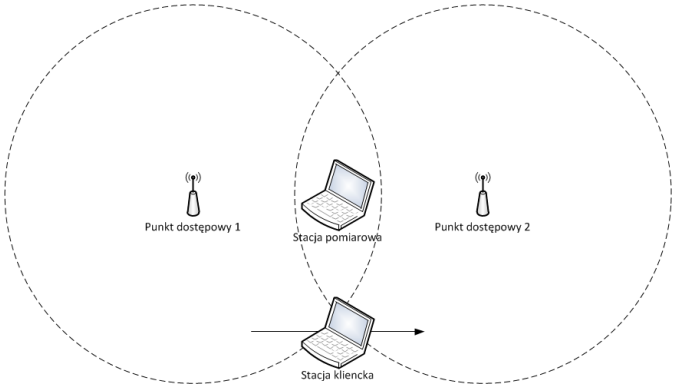
– Punkt dostępowy #2 dokonuje asocjacji stacji klienckiej i potwierdza to zdarzenie wysyłając ramkę typu *Association response*.

Łatwo zauważyć, że zjawisko roamingu jest kluczowe w przypadku systemu czasu rzeczywistego zarządzającego stacjami mobilnymi na rozległym obszarze (2). System może wykorzystywać wiele punktów dostępowych, które obsługuje poprzez sieć przewodową (ang. *Ethernet*). Każda zarządzana stacja w trybie AP przystosowana jest do działania w panujących na swoim obszarze warunkach zaszumienia łącza. Roaming 802.11 byłby w tym wypadku główną przyczyną przełączania kanału radiowego interfejsu NIC w mobilnych stacjach klienckich.



Rysunek 2: System z mobilnym agentem

Oczywiście roaming nie implikuje ruchu żadnej z maszyn, co ułatwia przeprowadzenie pomiaru. Wystarczy doprowadzić do sytuacji, w której moc sygnału punktu dostępowego spadnie poniżej progu (ang. *roaming treshold*), który powoduje decyzję o rozwiązaniu asocjacji stacji klienckiej.



Rysunek 3: Roaming 802.11: Środowisko pomiarowe.

Środowisko pomiarowe

W skład środowiska pomiarowego (3) wchodzą dwa punkty dostępowe, stacja kliencka oraz stacja pomiarowa. Punkty dostępowe pracują na różnych częstotliwościach. Do wyboru, zgodnie ze standardem 802.11g ([2]), są kanały numer 1, 5, 9, lub 13. Są to nienachodzące na siebie zakresy częstotliwości. W celu ułatwienia roamingu stacja kliencka umieszczona jest na granicy zasięgu punktów dostępowych. Stacja pomiarowa musi znajdować się w zasięgu stacji klienckiej, oraz obydwu punktów dostępowych (musi być w stanie rejestrować ruch sieciowy). Należy zwrócić uwagę na zapewnienie odpowiedniej jakości medium transmisyjnego. Wysoki poziom zakłóceń na kanałach wykorzystywanych w eksperymencie wprowadzi zakłamania, jeśli interesuje nas wyłącznie czas trwania samej procedury roamingu.

Mierzona wartość: Czas roamingu

Czas roamingu 802.11 rozumiem jako czas (4) mierzony od momentu decyzji stacji klienckiej o zaprzestaniu normalnej wymiany danych z punktem dostępowym do momentu powiązania z nową stacją w trybie AP o mocniejszym sygnale. Zdarzeniem inicjującym pomiar jest wysłanie przez stację kliencką pierwszej ramki rozwiązującej asocjację (ang. *Disassociation frame*). Pomiar zostaje zakończony w momencie wysłania przez nowy punkt dostępowy ramki potwierdzającej asocjację nowej stacji (ang. *Association Response frame*).

Podstawowa procedura pomiarowa składa się z następujących kroków:

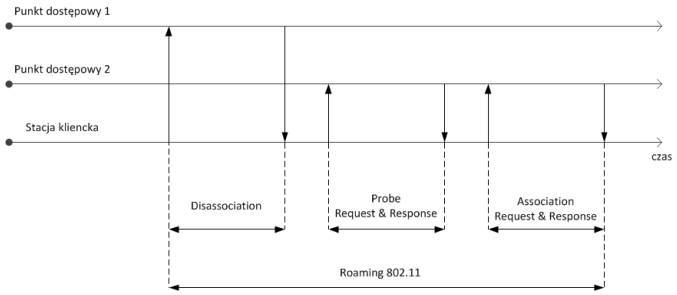
– Stacja kliencka przeprowadza asocjację z punktem dostępowym 1.

– Stacja kliencka przemieszcza się poza zasięg punktu dostępowego 1 i wykonuje procedurę roamingu do punktu dostępowego 2.

– Stacja pomiarowa wykrywa próbę rozwiązania asocjacji i rozpoczyna pomiar czasu.

– Punkt dostępowy 2 dokonuje asocjacji stacji klienckiej.

– Stacja pomiarowa rejestruje potwierdzenie asocjacji stacji klienckiej i zatrzymuje pomiar czasu.



Rysunek 4: Roaming 802.11: Czas roamingu.

Wnioski

0.4 Obsługa 802.11 w systemie operacyjnym Linux

0.4.1 Sterowniki kart radiowych

0.4.2 Interfejs wywołań systemowych

0.4.3 Warstwa pośrednia *mac80211*

0.4.4 Interfejs oparty na gniazdach *nl80211*

0.5 Narzędzie pomiarowe: hop-sniffer

Niniejszy rozdział poświęcony jest opisowi aplikacji powstałej na podstawie wymagań sformułowanych w opisie pomiaru (link). Analiza wymagań podyktowała stworzenie programu, który umożliwiałby pogląd ramek zarządzających komunikacją w standardzie 802.11 (ang. *Management frames*) oraz analizę zależności czasowych między nimi.

Wymagane okazało się stworzenie aplikacji nasłuchującej (ang. *sniffer*) przystosowanej do obserwacji typowych scenariuszy zachodzących w komunikacji w medium bezprzewodowym. Przystosowanie to rozumiem jako możliwość konfiguracji programu pod kątem wybranego zjawiska i środowiska pomiarowego.

0.5.1 Środowisko pracy programu.

Program hop-sniffer został przygotowany dla systemu operacyjnego Linux w wersji jądra 2.6. W wyborze systemu operacyjnego kierowałem się głównie metodą implementacji sterowników urządzeń bezprzewodowych i obsługującej je warstwy pośredniej jądra.

System Linux był wyborem oczywistym ze względu na możliwość konfiguracji interfejsów NIC w sposób umożliwiający przetwarzanie ramek typu MGMT (ang. *management*) standardu 802.11 za pomocą aplikacji w przestrzeni użytkownika.

Kolejną zaletą wybranego systemu jest możliwość konfiguracji najbardziej odpowiedniej dystrybucji i kompilacji powstałego rozwiązania jedynie z użyciem opcji dedykowanych dla aplikacji pomiarowej. W tym wypadku najbardziej pożądane jest minimalistyczne środowisko, które w możliwie najmniejszym stopniu wpływało będzie na prezentowane przez program wyniki pomiarów. Jako środowisko zalecane wybrałem system Arch Linux (link).

Biorąc pod uwagę program komunikujący się z kartą radiową w systemie Linux należy zwrócić szczególną uwagę na kwestię sterowników. Od sterowników urządzeń bezprzewodowych zależy jakie polecenia i tryby pracy interfejsów będą dostępne do konfiguracji w przestrzeni użytkownika. Ze względu na aktualne dążenie programistów jądra do unifikacji interfejsu obsługi urządzeń standardu 802.11 powstała warstwa pośrednia *mac80211*. Postanowiłem oprzeć aplikację hop-sniffer o sterowniki działające w tej warstwie ze względu na wspólny, oparty na gniazdach interfejs komunikacyjny *nl80211*. Kluczowym wymaganiem stawianym sterownikowi jest implementacja polecenia umożliwiającego utworzenie wirtualnego interfejsu karty radiowej pracującego w trybie *monitor*.

Wprowadzenie karty radiowej w tryb *promiscuous* powoduje jedynie wyłączenie filtracji adresów MAC. Program hop-sniffer musi mieć możliwość odbierania ramek standardu 802.11 bez potrzeby asocjacji z SSID (ang. *Service Set Identifier*) żadnej sieci. Wyłączenie filtracji SSID możliwe jest jedynie w trybie *monitor*.

W swojej pracy skoncentrowałem się na współpracy ze sterownikiem *ath9k*. Jest to całkowicie otwarty sterownik do urządzeń standardu 802.11bgn firmy *Atheros*. Za wykorzystaniem sterownika przemawia dostępność wspieranych przez niego urządzeń, implementacja szerokiej gamy poleceń interfejsu *nl80211* oraz możliwość pracy w trybie *monitor*.

0.5.2 Biblioteki programistyczne.

Implementacja aplikacji pomiarowej wymagała zastosowania API (ang. *Application interface*) umożliwiającego pochwycenie ramek zarządzających komunikacją 802.11 w przestrzeni użytkownika. Podczas procesu tworzenia programu hop-sniffer rozpatrzyłem zastosowanie dwóch bibliotek: *libnl* i *libp-*

cap.

Nasłuchiwanie za pomocą interfejsu *nl80211*.

Libnl jest to API (ang. *Application interface*) służące do komunikacji między przestrzenią użytkownika i warstwą *mac80211* jądra systemu operacyjnego. Interfejs *nl80211* tej warstwy oparty jest o system gniazd *Generic Netlink* (w odróżnieniu od stosowanych dawniej wywołań systemowych *IOCTL*).

Warstwa pośrednia definiuje rodzinę gniazd (ang. *Generic netlink family*) oraz rejestruje w jej obrębie zestaw poleceń w postaci akceptowanych rodzajów wiadomości. Sterowniki urządzeń 802.11 implementują powyższy interfejs poprzez inicjalizację odpowiadających poleceniom wskaźników na funkcje własnymi operacjami. Każda wiadomość akceptowana przez daną rodzinę posiada własną nazwę oraz wskaźnik na strukturę określającą ilość i typy atrybutów (ang. *Generic netlink attribute policy*), która pełni funkcję kontroli poprawności. Struktura ta zwana *nla_policy* stanowi wytyczne co do sposobu konstrukcji skierowanego do jądra polecenia oraz ekstrakcji danych z odebranej wiadomości.

W wyniku analizy dokumentacji uznałem, że możliwa będzie implementacja programu nasłuchującego z wykorzystaniem następujących mechanizmów udostępnianych przez interfejs *nl80211*:

- Grupowych adresów (ang. *Multicast groups*) odbiorców wiadomości.
- Komendy *NL80211_CMD_REGISTER_FRAME*.
- Własnych funkcji obsługi zdarzeń (ang. *Custom callback*).
- Komendy *NL80211_CMD_FRAME*.

Adresy grupowe są wykorzystywane przez jądro do rozgłaszania zdarzeń warstwy *mac80211* do zainteresowanych procesów (posiadających gniazdo z członkostwem w danej grupie rozgłaszania). W celu otrzymywania wszystkich zdarzeń należy zarejestrować gniazdo (1) we wszystkich czterech grupach: *Configuration, Scan, Regulatory i MLME*. Podczas rejestracji (1) funkcja *nl_get_multicast_id(3)* przyjmuje uchwyt gniazda komunikacyjnego, nazwę rodziny, do której odnosi się zapytanie i nazwę grupy, której identyfikator chcę uzyskać. W wyniku wywołania otrzymuję liczbę całkowitą, którą mogę wykorzystać w celu rejestracji danego gniazda w grupie rozgłaszania przy pomocy funkcji *nl_socket_add_membership(2)*.

Listing 1: Przykład rejestracji gniazda w grupie *Configuration*.

```
1 /* Get configuration multicast group ID */
  multicast_id = nl_get_multicast_id(cd->nl_sock,
3         "nl80211", "config");
```

```
if (multicast_id < 0)
        return multicast_id;

/* Add membership to configuration multicast group */
ret = nl_socket_add_membership(cd->nl_sock, multicast_id);
if (ret)
        return ret;
```

Komenda *NL80211_CMD_REGISTER_FRAME* pozwala na rejestrację wybranych typów ramek do przetwarzania w przestrzeni użytkownika. Wymagane atrybuty to indeks interfejsu radiowego (*NL80211_ATTR_IFINDEX* to liczba całkowita 32-bitowa), typ ramki (*NL80211_ATTR_FRAME_TYPE* to liczba całkowita 16-bitowa) oraz wzorca zawierającego pierwsze bajty ramki (*NL80211_ATTR_FRAME_MATCH* to wzorzec binarny z podaną długością), które powinny być dopasowane. Należy wziąć pod uwagę fakt, że w tym wypadku aplikacja musi obsłużyć dany typ ramek, gdyż nie zostaną one odpowiednio przetworzone w jądrze. Zamknięcie gniazda komunikacyjnego za pomocą, którego dokonano rejestracji powoduje jej porzucenie.

W mojej aplikacji zgłaszanie ramek do obsługi przez program nasłuchujący jest częścią inicjalizacji. Należy zarejestrować wszelkie ramki niezbędne do obserwacji wybranego zjawiska. Proces budowania wiadomości (2) zaczyna się od stworzenia nagłówka opatrzonego odpowiednim adresem odbiorcy (identyfikatorem rodziny) oraz nazwą polecenia do wykonania. Służy do tego funkcja biblioteczna *genlmsg_put(8)*, która dodaje do otrzymanego uchwytu wiadomości nagłówek wybranej komendy przynależącej do identyfikatora podanej rodziny. Następnie dodaję atrybuty wymagane przez komendę specyfikując identyfikator (potrzebny w celu sprawdzenia poprawności) oraz wartość. Są one wstawiane do pól wiadomości za pomocą makr bibliotecznych *NLA_PUT* (odpowiadających typowi danych). Numer interfejsu tłumaczony jest z nazwy (np. *wlan0*) na indeks (typ całkowity). Rodzaj ramki to liczba całkowita 16-bitowa, którą w języku C możemy wprowadzić *in-situ* jako *0x0040* (ramka typu *Probe Request*). Po zbudowaniu prawidłowej wiadomości pozostaje wysłać ją za pomocą funkcji bibliotecznej *nl_send_auto_complete(2)*.

Listing 2: Przykład rejestracji ramki do obsługi w przestrzeni użytkownika.

```
/* Build netlink message header */
genlmsg_put(msg, 0, 0, genl_family_get_id(cd->nl80211),
        0, 0, NL80211_CMD_REGISTER_FRAME, 0);
/* Device interface index to use */
devid = if_nametoindex(if_name);
NLA_PUT_U32(msg, NL80211_ATTR_IFINDEX, devid);
/* Register frame type/subtype */
NLA_PUT_U16(msg, NL80211_ATTR_FRAME_TYPE, fr_type);
/* Frame match for MGMT frames is NULL */
NLA_PUT(msg, NL80211_ATTR_FRAME_MATCH, 0, NULL);
/* Send message */
error = nl_send_auto_complete(cd->nl_sock, msg);
```

Po przyłączeniu gniazda do odpowiednich grup rozgłaszania i wybraniu niezbędnych typów ramek do przetwarzania przez program pozostaje rozpocząć nasłuchiwanie zdarzeń interfejsu *nl80211* (3). W mojej aplikacji wybór badanego zjawiska (sposobu reakcji na zdarzenia) zależy od rodzaju funkcji do której wskaźnik jest przekazywany podczas rozpoczęcia nasłuchu. Funkcja ta przekazywana jest jako argument procedury typu *callback* używanej do przetwarzania odebranych wiadomości (zdarzeń).

Zdefiniowana przeze mnie funkcja *custom_event_handler* zostaje wybrana do obsługi zdarzeń za pomocą procedury bibliotecznej *nl_cb_set(5)* z flagami *NL_CB_VALID* (używana do wiadomości poprawnych) i *NL_CB_CUSTOM* (zdefiniowana przez użytkownika). Dodatkowo przekazuję wskaźnik na stworzone przez siebie argumenty wywołania (w tym uchwyt do funkcji obsługującej obserwację wybranego zjawiska *fptr_handle_frame*), które będą dostępne w bloku procedury *custom_event_handler*. Blokująca procedura biblioteczna *nl_recvmsgs* oczekuje na zdarzenia i wywołuje dla nich przekazaną jej funkcję *callback*.

Listing 3: Fragment kodu procedury rozpoczynającej obsługę zdarzeń.

```
/* Choose scenario type. */
args.handle_frame = fptr_handle_frame;
/* ... */
/* set custom event handler and pass arguments to it. */
nl_cb_set(cb, NL_CB_VALID, NL_CB_CUSTOM,
        custom_event_handler, &args);
/* ... */
/* Listen events. */
while (!command)
{
        nl_recvmsgs(cd->nl_sock, cb);
}
```

Komenda *NL80211_CMD_FRAME* służy do nadawania i odbierania wybranych typów ramek z poziomu aplikacji użytkownika. W przypadku programu nasłuchującego interesuje mnie funkcjonowanie tej wiadomości jako zdarzenia propagowanego przez jądro w sytuacji otrzymania nieobsłużonej ramki.

Odebranie przez nasłuchujące gniazdo poprawnej wiadomości interfejsu *nl80211* powoduje wywołanie własnej funkcji obsługi *custom_event_handler* (4). Celem jest rozpoznanie komendy *NL80211_CMD_FRAME* i przekazanie jej atrybutu *NL80211_ATTR_FRAME* do funkcji obsługujące obserwowane zjawisko. Atrybut *NL80211_ATTR_FRAME* reprezentuje odebraną ramkę (nagłówek i pole danych) i jest typu binarnego (ciąg bajtów).

Pierwszym krokiem jest ekstrakcja nagłówka wiadomości *netlink* w postaci argumentu wywołania i jego rozpakowanie do postaci struktury *genlmsghdr* za pomocą funkcji bibliotecznych *nlmsg_hdr(1)* i *nlmsg_data(1)*. Struk-

tura ta zawiera pole *cmd* będące identyfikatorem komendy, którego używam w bloku *switch*.

Niezbędna jest ekstrakcja atrybutów wiadomości za pomocą procedury bibliotecznej *nla_parse(5)*, która otrzymuje bufor na atrybuty (mogący pomieścić *NL80211_ATTR_MAX* + 1 atrybutów), stałą biblioteczną oznaczającą liczbę wszystkich atrybutów *NL80211_ATTR_MAX*, początek listy atrybutów (struktura *nlattr*) znaleziony za pomocą funkcji bibliotecznej *genlmsg_attrdata(2)* i długość listy atrybutów otrzymana z wykorzystaniem funkcji *genlmsg_attrlen(2)*. Otrzymaną w ten sposób tablicę struktur *nlattr* indeksuję identyfikatorem atrybutu *NL80211_ATTR_FRAME* przekazuję go jako argument wywołania funkcji obsługującej obserwowane zjawisko *handle_frame* przekazanej w zdefiniowanej wcześniej strukturze argumentów użytkownika *event_handler_args*.

Listing 4: Własna funkcja obsługi zdarzeń.

```
int custom_event_handler(struct nl_msg *msg, void *arg)
{
    /* Generic netlink message header */
    struct genlmsghdr *gnlh = nlmsg_data(nlmsg_hdr(msg));
    /* Buffer for attributes from netlink message */
    struct nlattr *msg_attr_buff[NL80211_ATTR_MAX + 1];
    struct event_handler_args *args = arg;

    /* Extract attributes */
    nla_parse(msg_attr_buff,
              NL80211_ATTR_MAX,
              genlmsg_attrdata(gnlh, 0),
              genlmsg_attrlen(gnlh, 0),
              NULL);

    /* Handle event according to type */
    switch (gnlh->cmd)
    {
    /* ... */
    case NL80211_CMD_FRAME:
        if(msg_attr_buff[NL80211_ATTR_FRAME])
            args->handle_frame(
                msg_attr_buff[NL80211_ATTR_FRAME]);
        break;
    /* ... */
    }
/* ... */
}
```

Przykładowym sposobem obsługi wybranego zjawiska komunikacji bezprzewodowej w standardzie 802.11 jest procedura *handle_frame* (5). Przekazanie funkcji obsługi poprzez wskaźnik jest sposobem na różnicowanie działania programu w zależności od scenariuszy komunikacyjnych, które są obiektem badań oraz dostarczenie ujednoliconego interfejsu ich implementacji.

Głównym krokiem procedury jest ekstrakcja atrybutu *nl80211* reprezentującego ramkę standardu 802.11 do postaci ciągu bajtów za pomocą funkcji bibliotecznej *nla_data(1)*. Otrzymana w ten sposób tablica jest indeksowana w poszukiwaniu konkretnych bajtów, a ekstrakcja informacji polega na zastosowaniu maski bitowej (przykładowo *0xfc* do bajtu podtypu).

Listing 5: Funkcja *handle_frame*.

```
void handle_frame(struct nlattr *nl_frame)
{
        uint8_t *frame;
        /* ... */
        /* Extract frame byte array from netlink attribute */
        frame = nla_data(nl_frame);
        /* ... */
        switch (frame[0] & 0xfc)
        {
                case 0x10: /* assoc resp */
                        /* ... */
                        break;
                case 0xa0: /* disassoc */
                        /* ... */
                        break;
        }
}
```

Stworzona przeze mnie aplikacja oparta na powyżej opisanej metodyce spełniała założenia powstałe w fazie analizy wymagań dla testowanych ramek standardu 802.11 typu *Probe Request*. Niestety rejestracja ramek dla interfejsu typu *monitor* okazała się niemożliwa, a typy ramek możliwe do odbierania na poszczególnych interfejsach (ang. *Supported RX frame types*) są całkowicie zależne od implementacji sterownika i mocno ograniczone ze względu na jego typ. Aktualnie typy ramek 802.11 możliwe do wysyłania i dobierania na danym interfejsie są dostępne i ogłaszane w atrybutach wirtualnego urządzenia reprezentującego kartę radiową (ang. *Wiphy*). Urządzenie to jest zaimplementowane w warstwie pośredniej *mac80211*, a struktury je opisujące wypełniane są przez odpowiadający mu sterownik.

Inspekcja dostępnych w przestrzeni użytkownika ramek możliwa jest dzięki analizie odpowiedzi interfejsu *nl80211* na komendę *NL80211_CMD_GET_WIPHY* z dodatkową flagą nagłówka *netlink* o nazwie *NLM_F_DUMP*, która powoduje przekazanie do wysyłającej aplikacji wiadomości ze wszystkimi parametrami wybranych urządzeń *Wiphy*.

Listing 6: Część atrybutów *Wiphy* o identyfikatorze *phy0* (program *iw-3.2*).

```
marcin@marcin-PC:~/iw-3.2$ ./iw phy0 info
Wiphy phy0
        ...
        Supported RX frame types:
                * IBSS: 0x00d0
                * managed: 0x0040 0x00d0
```

```
7                   * AP: 0x0000 0x0020 0x0040 0x00a0 0x00b0
                      0x00c0 0x00d0
9                   * AP/VLAN: 0x0000 0x0020 0x0040 0x00a0
                      0x00b0 0x00c0 0x00d0
11                  * mesh point: 0x00b0 0x00c0 0x00d0
                    * P2P-client: 0x0040 0x00d0
13                  * P2P-GO: 0x0000 0x0020 0x0040 0x00a0
                      0x00b0 0x00c0 0x00d0
15          ...
```

Analiza dostępnych do odebrania ramek wskazuje, że nie jest możliwe badanie niektórych zjawisk (np. roaming 802.11). Interfejsy nie pozwalają na rejestrację w jądrze ramek typu *Association Response* (identyfikator *0x1*), a odbieranie ramek typu *Disassociation* (identyfikator 0xA) wymaga wprowadzenia interfejsu w tryb *Master* (uruchomienia programu *hostapd*, a więc utworzenia na komputerze punktu dostępowego).

Oczywiście, jeśli nasłuchiwanie nie będzie prowadzone w trybie *monitor* to program i tak nie otrzyma ramek z sieci, której nie jest członkiem (ze względu na filtrację SSID).

Powyższe problemy powodują, że mimo uniwersalności i licznych zalet związanych z prostym sposobem ekstrakcji danych z ramek standardu 802.11 biblioteka *libnl* nie nadaje się do zastosowania w aplikacji opisanej wymaganiami sformułowanymi w fazie opisu procedury pomiarowej (link).

Nasłuchiwanie za pomocą biblioteki typu *pcap*.

Biblioteka *libpcap* (ang. *Packet capture library*) udostępnia wysokopoziomowy interfejs przechwytywania pakietów (również tych, które nie są kierowane do danej maszyny) co czyni ją odpowiednim narzędziem do implementacji programu hop-sniffer.

Procedura inicjalizacji programu nasłuchującego wymaga podjęcia następujących kroków:

1. Przygotowanie wirtualnego interfejsu pomiarowego w trybie *monitor*.
2. Utworzenia urządzenia przechwytującego.
3. Ustalenie długości migawki.
4. Wprowadzenie interfejsu radiowego w tryb *promiscuous*.
5. Ustalenie niedoczasu dla urządzenia przechwytującego.
6. Aktywacja urządzenia przechwytującego.
7. Sprawdzenie długości migawki.
8. Sprawdzenie przynależności do sieci.

9. Kompilacja kodu filtra pakietów.
10. Ustalenie skompilowanego filtra w urządzeniu przechwytującym.
11. Uruchomienie pętli głównej programu.

Przygotowanie interfejsu pomiarowego może być wykonane poza programem za pomocą narzędzia konfiguracji interfejsów *iw* (link). Interfejs w trybie *monitor* tworzy się (7) poprzez podanie nazwy istniejącego interfejsu radiowego, dzięki czemu możliwa jest identyfikacja urządzenia *Wiphy*, które ma być współdzielone.

Listing 7: Dodanie interfejsu *mon0* w trybie *monitor*

```
1 marcin@marcin-PC:~$ iw dev wlan0 interface add mon0 type monitor
```

Podstawowym krokiem programu jest utworzenie uchwytu do urządzenia przechwytującego. Zadanie to polega na inicjalizacji wskaźnika na strukturę *pcap_t*. Struktura ta zdefiniowana jest w bibliotece w sposób nietransparentny (ang. *opaque structure*), więc jej zawartość deklarowana jest w plikach źródłowych, a nie nagłówkowych. Funkcja tworząca urządzenie (8) przyjmuje tablicę znaków określającą nazwę interfejsu oraz tablicę bajtów przeznaczoną na kody ewentualnych błędów.

Listing 8: Utworzenie uchwytu urządzenia przechwytującego

```
1 /* Create capture device */
pdev = pcap_create(device, ebuf);
```

Pierwszą z ważnych do ustalenia opcji jest długość migawki (ang *snapshot length*). Długość wystarczająca do przechwycenia całej ramki wynosi *65000* (9). Zdecydowałem się na przechwytywanie całych ramek ze względu na przyszły rozwój aplikacji. Aktualnie nie jest możliwe do ustalenia do jakich typów pakietów będzie używany program. Rozmiar ramek zmienia się w zależności od użytych metod szyfrowania oraz zawartości nagłówka *radiotap*. Użyta funkcja biblioteczna *pcap_set_snaplen(2)* przyjmuje uchwyt do urządzenia oraz zmienną typu *long long* reprezentującą długość migawki.

Następnie należy wprowadzić urządzenie w tryb *promiscuous* i ustalić niedoczas dla przechwytywania. Niedoczas określa odstępy w jakich biblioteka będzie dokonywała odczytów z urządzenia nasłuchującego. Nie jest to parametr, który zakłóca pomiar, gdyż czas otrzymania ramki odczytywany jest ze znacznika w nagłówku *pcap*, a nie naliczany w aplikacji pomiarowej. Wartość 1000 milisekund pozwala na równomierne czytanie z bufora pakietów. Ustawień dokonuje się w sposób analogiczny podając uchwyt urządzenia przy wywołaniu funkcji *pcap_set_promisc(2)* z wartością 1 (aby ustawić tryb *promiscuous*) lub *pcap_set_timeout(2)* z wartością 1000 (aby ustawić niedoczas 1000 milisekund).

Listing 9: Inicjalizacja parametrów urządzenia przechwytującego.

```
/* Init capture device */
/* Set snapshot length */
err = pcap_set_snaplen(pdev, snapshot_size);
/* ... */
/* Set promiscus mode */
err = pcap_set_promisc(pdev, 1);
/* ... */
/* Set timeout */
err = pcap_set_timeout(pdev, 1000);
```

Na zakończenie procesu inicjalizacji uchwytu urządzenia należy go aktywować (10). Jest to okazja do obsługi wszelkich ostrzeżeń i błędów wygenerowanych w wyniku ustawionych powyżej opcji. Pomyślna aktywacja pozwala na rozpoczęcie nasłuchu w medium pracy wybranego interfejsu. Do błędów zaliczam wszelkie sytuacje, które nie pozwolą na dalszą poprawną pracę programu, a więc następujące wartości zwracane:

- **PCAP_ERROR_NO_SUCH_DEVICE:** Podana nazwa interfejsu nie istnieje.
- **PCAP_ERROR_PERM_DENIED:** Użytkownik wywołujący program nie posiada uprawnień do otwarcia wybranego interfejsu.
- **PCAP_ERROR:** Błąd, który nie jest zdefiniowany w nagłówku biblioteki.

Ostrzeżenia pozwalają na dalszą pracę programu, ale mogą poważnie ograniczyć jego funkcjonalność:

- **PCAP_WARNING_PROMISC_NOTSUP:** Tryb interfejsu *promiscuous* nie jest wspierany przez dostępne urządzenie. Będzie miała miejsce filtracja adresów MAC.
- **PCAP_WARNING:** Ostrzeżenie, które nie jest zdefiniowane w nagłówku biblioteki.

Listing 10: Aktywacja urządzenia przechwytującego.

```
/* Activate capture device */
err = pcap_activate(pdev);
```

Po uruchomieniu urządzenia nasłuchującego należy sprawdzić część parametrów związanych z aktywacją (11). Po pierwsze rozmiar migawki, ponieważ mógł on ulec zmianie. Następnie fakt przynależności wybranego interfejsu przechwytywania do sieci. Rozmiar migawki pobiera się wykorzystując funkcję *pcap_snapshot(1)* podając uchwyt urządzenia. Podglądu sieci dokonuję wywołując procedurę *pcap_lookupnet(4)* z argumentem nazwy interfejsu, wskazania na 32-bitowe liczby całkowite reprezentujące sieć i jej

maskę (do wypełnienia przez wywołanie) oraz tablicę bajtów na kod ewentualnych błędów. Kroki te są wykonywane w celach prezentacji informacji i mogą wygenerować jedynie ostrzeżenia.

Listing 11: Sprawdzenie parametrów po aktywacji urządzenia.

```
/* Check snapshot size after init */
i = pcap_snapshot(pdev);
/* ... */
/* Check sniffed network */
if (pcap_lookupnet(device, &localnet, &netmask, ebuf) < 0)
{
/* ... */
}
```

Biblioteka *libpcap* udostępnia kompilator reguł logicznych opisu filtrów na język BPF (ang. *Berkley packet filter*). Jest on interesujący z perspektywy mojego programu ze względu na potrzebę maksymalnej redukcji opóźnień. Wprowadzenie instrukcji warunkowych do kodu programu w celu filtracji pakietów gdy jądro i tak przekazuje wszystkie przechwycone pakiety jest bardzo nieefektywne.

Jądro systemów operacyjnych z pod znaku BSD (ang. *Berkley Software Distribution* posiada wbudowany mechanizm szybkiej filtracji dostępny w postaci urządzeń */dev/bpf0*, */dev/bpf1* itd. Zezwalają one na powiązanie zdefiniowanego przez użytkownika filtra pakietów. Asocjacja deskryptora urządzenia *bpf* z otwartym gniazdem powoduje wpływ jego reguł filtrujących na odbierane ramki. Zgodnie z dokumentacją można oczekiwać, że podobny mechanizm znajdzie się w wersji jądra Linux 3.0.

Linux w wersji 2.6 oferuje jednak wystarczająco wydajny mechanizm zwany LSF (ang. *Linux Socket Filter*), który akceptuje język BPF. Jest to wyjątkowo ważne, gdyż jego brak wymusza filtrację pakietów wewnątrz biblioteki *pcap*, a więc poza jądrem co negatywnie wpływa na efektywność rozwiązania.

Maszyna stanowa LSF może zostać uruchomiona zaraz po odebraniu pakietu ze sterownika urządzenia. Filtracja odbywa się wewnątrz procedur protokołu PF_PACKET używanego podczas nasłuchiwania. Protokół ten pomija standardowy przepływ danych przez stos TCP/IP i pozwala na bezpośrednie odebranie ramki z kompletem nagłówków w postaci surowej wykorzystując gniazdo typu SOCK_RAW.

Kompilacja kodu filtra w bibliotece *libpcap* odbywa się poprzez wywołanie funkcji *pcap_compile(5)* podając uchwyt urządzenia, wskazanie na strukturę *bpf_program*, która zostanie wypełniona utworzonym filtrem, ciąg znaków zawierający opis filtra za pomocą języka reguł logicznych, przełącznik optymalizacji oraz maskę sieci, w której aplikacja prowadzi nasłuch. W przypadku mojej aplikacji nasłuchującej maska sieci, w większości przypadków, nie będzie znana (będzie miała wartość *PCAP_NETMASK_UNKNOWN*).

Zakończenie procesu ustalania filtra odbywa się za pomocą funkcji bibliotecznej *pcap_setfilter(2)* podając uchwyt urządzenia i wskazanie na jego program (12).

Listing 12: Kompilacja i ustalenie programu filtra BPF

```
/* Compile filter code */
if (pcap_compile(pdev, &filtercode, filter, 0, netmask) < 0)
/* ... */
/* Set compiled filter */
if (pcap_setfilter(pdev, &filtercode) < 0)
/* ... */
```

Ostatecznym krokiem programu jest wystartowanie pętli głównej programu, która będzie wywoływać własną funkcję obsługi pakietów. Rozpoczynam od podłączenia procedury *handle_packet* do wskaźnika na funkcję *callback*. Do procedury bibliotecznej *pcap_loop(4)* przekazuję uchwyt urządzenia, liczbę pakietów po jakiej ma się zatrzymać (-1 oznacza nieskończoność), wskaźnik na funkcję obsługi i własną strukturę argumentów użytkownika.

Funkcja *handle_packet* otrzymuje na wejściu strukturę użytkownika, nagłówek pakietu *pcap_pkthdr* (standardowy nagłówek dodawany do każdego pakietu przez bibliotekę) oraz wskaźnik na tablicę bajtów zawierających migawkę odebranej ramki. Nasłuchiwanie na interfejsie typu *monitor* wiąże się z faktem otrzymywania przez bibliotekę *libpcap* nagłówka typu *radiotap*, więc funkcja rozpoczyna od jego przetwarzania.

Listing 13: Wystartowanie pętli głównej programu.

```
 /* Prepare arguments for loop */
 callback = handle_packet;
 /* ... */
err = pcap_loop(pdev, cnt, callback, pcap_largs);
```

Listing 14: Procedura przetwarzania ramek.

```
static void
handle_packet(u_char *user, const struct pcap_pkthdr *h,
        const u_char *sp)
{
        u_int hdrlen;
        /* ... */
        hdrlen = if_radiotap_parse(h, sp);
        /* ... */
}
```

Mechanizm przechwytywania ramek oferowany przez bibliotekę *libpcap* jest wystarczający do implementacji programu spełniającego wymagania wypracowane w procesie projektowania procedury pomiarowej (link). W przeciwieństwie do biblioteki *libnl* możliwe jest odbieranie dowolnego rodzaju pakietów standardu 802.11 bez potrzeby asocjacji z żadnym punktem

dostępowym (uczestnictwa w sieci), a zatem z minimalną ingerencją w środowisko pomiarowe. Jego użycie wymaga jednak bardziej skomplikowanych metod ekstrakcji danych, ze względu na konieczność odczytywania nagłówka *radiotap*. Krok ten jest niezbędny ze względu na fakt wstawiania przez niektóre karty radiowe dodatkowego odstępu między nagłówkiem, a pozostałą częścią ramki (ang. *Atheros padding*). Informacja o tym dostępna jest w postaci pola *radiotap*, które trzeba odnaleźć.

0.5.3 Implementacja programu *hop-sniffer*.

Niniejszy rozdział opisuje implementację programu *hop-sniffer*. Pomijam część opisu związaną z inicjalizacją urządzenia przechwytującego i pętli obsługującej wywoływanie funkcji *callback* (kroki te zostały objaśnione w rozdziale dotyczącym bibliotek programistycznych (link)). Aplikacja została stworzona w języku *C* pod kątem wybranego wcześniej środowiska Linux. Biblioteka *libpcap* wykorzystywana jest do przechwytywania i wstępnej selekcji pakietów. Używany do selekcji filtr BPF zapisany jest w postaci reguł logicznych w pliku. Program nasłuchujący odczytuje plik jako ciąg znaków, kompiluje go i ustawia jego program jako nowy filtr.

Podstawą logiki programu (8) jest reagowanie na przetworzone składniki pakietów w celu pomiaru opóźnienia roamingu stacji klienckiej w standardzie 802.11.

Obsługa sygnałów i zwalnianie zasobów.

Głównie ze względu na fakt, iż funkcja startująca pętlę przetwarzania pakietów *pcap_loop(4)* działa w sposób blokujący istnieje potrzeba obsługi sygnałów. Procedury obsługi są możliwie najkrótsze i skupiają się na prawidłowym zwolnieniu zasobów w przypadku otrzymania określonego typu sygnału.

Sygnały obsługiwane przez aplikację *hop-sniffer* to:

- SIGPIPE
- SIGTERM
- SIGINT
- SIGCHLD
- SIGHUP

Otrzymanie sygnałów SIGPIPE, SIGTERM, SIGINT powoduje przerwanie pętli przechwytywania pakietów (zwolnienie zasobów urządzenia przechwytującego) i zwolnienie uchwytu do gniazda używanego do wywoływania poleceń IOCTL.

Sygnał SIGCHLD wstrzymuje daną instancję procesu programu *hop-sniffer* w oczekiwaniu na zakończenie pracy jego procesów potomnych.

Sygnał SIGHUP wspiera możliwość pracy programu w tle (po wylogowaniu wywołującego użytkownika) przyporządkowując sygnałowi procedurę zwalniania zasobów jedynie, gdy nie jest w użyciu program *nohup(1)*.

Pomiar zależności czasowych między ramkami.

Wszystkie pakiety przechwytywane przez bibliotekę *libpcap* posiadają dodany wspólny nagłówek (15) (ang. *Generic Pcap Header*), który ujednolica ich obsługę na przestrzeni różnych interfejsów, z których mogą pochodzić. Jedną z części reprezentującej go struktury (*pcap_pkthdr*) jest pole *ts* typu *timeval*. Pole to jest stemplem czasowym momentu odebrania ramki i służy programowi *hop-sniffer* do wszelkich obliczeń związanych z opóźnieniami między poszczególnymi zdarzeniami.

Listing 15: Wspólny nagłówek pakietów *pcap*.

```
1 struct pcap_pkthdr {
        struct timeval ts;
3       bpf_u_int32 caplen;
        bpf_u_int32 len;
5 }
```

Struktura *timeval* (16) jest jedną z metod reprezentacji fragmentu czasu w systemie Linux. Jest ona automatycznie wypełniana w ramach działania biblioteki *libpcap*. Pole *tv_usec* zapewnia wystarczającą dokładność pomiaru ze względu na aktualny fakt nie występowania w standardzie 802.11 interwałów czasowych między zdarzeniami (wysyłanymi ramkami) poniżej rzędu milisekund.

Listing 16: Struktura *timeval*.

```
1 typedef struct timeval {
        long tv_sec;
3       long tv_usec;
} timeval;
```

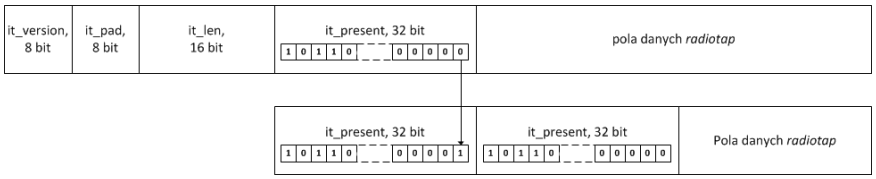
Biorąc pod uwagę fakt, że czas trwania dowolnego zdarzenia jest *de facto* opóźnieniem między dwoma pakietami (rozpoczynającym i kończącym wybrany scenariusz komunikacji) wystarczy programowo zapamiętywać stemple czasowe wybranych typów ramek. Obliczanie opóźnienia polega na wykonaniu różnicy wartości zapamiętanych czasów otrzymania ramki rozpoczynającej i kończącej obserwowane zjawisko.

Przetwarzanie nagłówka *radiotap*.

Odczytywanie danych z nagłówka *radiotap* jest konieczne ze względu na potrzebę sprawdzenia pola zawierającego flagi (ang. *radiotap flags*). Ma ono

postać bajtu danych będącego bitmapą ustawionych przełączników . Aplikacja *hop-sniffer* sprawdza czy ustawione są flagi 0x10 (*IEEE80211_RADIOTAP_F_FCS*) lub 0x20 (*IEEE80211_RADIOTAP_F_DATAPAD*) oznaczające kolejno dodatkowy odstęp między nagłówkiem i polem danych ramki 802.11 i fakt posiadania przez ramkę części FCS (ang. *Frame check sequence*). Są to informacje konieczne do prawidłowego przetworzenia pakietu standardu 802.11.

Pierwszym elementem nagłówka (5) jest pole *it_version* o rozmiarze 8 bitów reprezentujące wersję nagłówka. Aktualnie jest ono ustawiane na wartość 0. Element *it_pad* o rozmiarze 8 bitów nie jest aktualnie używany. Kolejnym polem jest *it_len* o rozmiarze 16 bitów wyznaczające długość całego fragmentu *radiotap* (wraz z danymi). Ostatnim i ważnym elementem nagłówka jest pole *it_present* będące 32-bitową mapą posiadanych przez daną ramkę pól danych *radiotap*. Bitmapa ta może być w prosty sposób poszerzana. Ustawienie ostatniego bitu (numer 31) oznacza, że dany element *it_present* poprzedza kolejną bitmapę. Obecność pola danych *radiotap flags* oznaczona jest poprzez ustawienie bitu numer 1.



Rysunek 5: Nagłówek *radiotap*.

Funkcja przetwarzająca aplikacji *hop-sniffer* otrzymuje na wejściu ramkę w postaci ciągu bajtów. Ciąg ten zostaje rzutowany na strukturę odpowiadającą wyżej opisanemu składowi nagłówka i odczytywane jest pole *it_len*. Odczytywanie danych z nagłówka wymaga ekstrakcji z formatu *little endian*, w którym są one kodowane. Parametr ten jest wykorzystywany do obliczenia wskaźnika na koniec nagłówka *radiotap*.

Pierwszym krokiem jest odnalezienie ostatniej bitmapy *it_present* sprawdzając jej 31 bit i ewentualnie przesuwając wskaźnik o 32 kolejne. Za bitmapami znajdują się pola przechowujące dane w naturalnym porządku binarnym (co 8, 16, 32 itd. bitów). Są one rozmieszczane według tego samego porządku co odzwierciedlające je numery wewnątrz map *it_present*.

Następnie program pomiarowy przegląda dostępne bitmapy i dla każdego kolejnego, ustawionego bitu rozpakowuje odpowiadające mu pole danych. Przeglądanie bitmapy od najmniej znaczącego bitu polega na odjęciu od niej liczby 1 i wykonania operacji XOR między tablicą wynikową i wejściową. W ten sposób program otrzymuje bitmapę z ustawionym jedynie aktualnie najmniej znaczącym bitem. W celu określenia odpowiadającego mu pola *radiotap* należy obliczyć na której pozycji w 32-bitowym słowie się on znajduje. Odnalezienie pozycji realizowane jest za pomocą zagnież-

dżonego makra, które wykonuje przesunięcia bitowe w prawo sprawdzając czy otrzymane słowo równe jest zeru. Przesunięcia wykonywane są kolejno połowiąc pozostałe do sprawdzenia słowo, czyli kolejno o 16, 8, 4 i 2 bity. Jeśli przesunięcie wyzerowało tablicę oznacza to, że bit znajduje się na numerze pozycji mniejszym niż przesunięcie i należy ponownie wywołać makro dla tej samej tablicy przesuwając o połowę mniej bitów. Otrzymanie słowa niezerowego oznacza, że numer bitu jest wyższy niż przesunięcie. W tym przypadku można zwrócić sumę liczby przesuniętych bitów (szukany numer jest większy) i wywołania makra przesuwającego o dwukrotnie mniejszą ilość bitów, ale dla aktualnego (już przesuniętego) słowa. Jest to implementacja wzorowana na algorytmie wyszukiwania binarnego opartego na idei *dziel i zwyciężaj* co sugeruje jej działanie w czasie logarytmicznym.

Specyfikacja *radiotap* zakłada, że programista znający nazwę pola zna również jego rozmiar. Pola danych rozmieszczone są zgodnie z naturalnym porządkiem binarnym. Oznacza to, że odczytując słowo określonej wielkości należy sprawdzić, czy mieści się ono w do tej pory przetworzonym fragmencie danych całkowitą liczbę razy. Jeśli tak nie jest to słowo należy odczytać z pozycji znajdującej się o brakującą liczbę bitów dalej, gdyż program trafił na wypełnienie (ang. *padding*). Jest to przyjęty standard konstrukcji nagłówka *radiotap*, więc *hop-sniffer* również go respektuje. Przykładowo (**??**), jeśli do tej pory program odczytał trzy pola o rozmiarze 8 bitów i otrzymuje polecenie odczytania słowa 16-bitowego to powinno być ono odczytane z adresu (uznając początek przestrzeni danych za 0) 32, a nie 24.

Do rozpakowywania pól danych nagłówka *radiotap* służy struktura *unpacker* (17). Pole *u_buf* inicjowane jest przez adresem pierwszego bajtu za ostatnią bitmapą *it_present*, *u_next* za pomocą wskaźnika na bajt za ostatnim odczytanym słowem, a *u_len* jest różnicą wskaźnika na początek ramki i ostatnią bitmapę.

Listing 17: Struktura *unpacker*.

```
struct unpacker {
        /** Pointer to the beginning of radiotap data fields area of packet. */
        u_int8_t                                        *u_buf;
        /** Pointer to the next packet area that was not yet extracted */
        u_int8_t                                        *u_next;
        /** Length of the radiotap data fields area */
        size_t                                           u_len;
};
```

Po odczytaniu każdego słowa zgodnie z wyżej opisanymi regułami wskaźnik *u_next* przenoszony jest do przodu o jego długość. Wskaźnik na początek fragmentu danych służy do obliczania ewentualnych wypełnień (ang. *padding*), a długość fragmentu do kontroli poprawności (jako warunek zatrzymujący przetwarzanie).

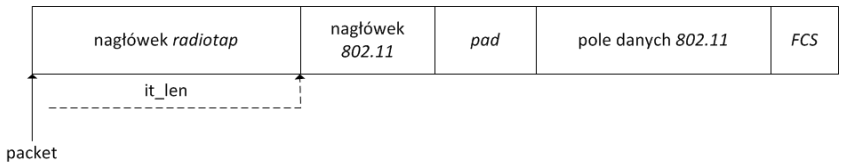
Odczytanie z bitmapy ustawionego bitu na pozycji 1 oznacza atrybut *IEEE80211_RADIOTAP_FLAGS*, który zgodnie ze specyfikacją ma 8 bitów.

Po rozpakowaniu jest on porównywany z maską *0x10* (właściwość oznaczająca dodatkowy odstęp za nagłówkiem 802.11) i *0x20* (ramka posiada fragment FCS). W pierwszym przypadku program ustawia zmienną *pad* na wartość 1, w drugim zmienną *fcslen* na wartość 4. Obydwie zmienne przekazywane są do procedury przetwarzania ramki standardu 802.11, gdzie będą potrzebne.

Nagłówek *radiotap* przechowuje również szczegółowe informacje dotyczące częstotliwości i mocy nadawania interfejsu przez, który został utworzony. Właściwość ta nie ma zastosowania w pomiarze roamingu 802.11, ale może być przydatna w bardziej skomplikowanych scenariuszach. Przetwarzanie nagłówka udostępnia więcej *meta-danych* dotyczących kanału komunikacyjnego co stanowczo przemawia na jego korzyść.

Przetwarzanie nagłówka standardu *802.11*.

Wskaźnik *packet* na początek migawki przechwyconej ramki (6) trafia na wejście funkcji przetwarzania nagłówka standardu 802.11 po przesunięciu o *it_len* bitów w przód. Tym samym powinien wskazywać na początek elementu FC (ang. *Frame control*). Do funkcji przekazane zostają również wartości zmiennych *pad* i *fcslen* wyliczone w procesie przetwarzania nagłówka *radiotap*.



Rysunek 6: Przesunięcie wskaźnika na początek ramki poza nagłówek *radiotap*.

Na podstawie danych uzyskanych z pakietu na tym etapie podejmowane są główne kroki procedury pomiarowej. Scenariusz pomiaru można rozumieć jako globalną strukturę, której pola wypełniane są w zależności od przepływu programu w wyniku wykrytego zdarzenia. Część struktur jest globalna, gdyż jest to bardziej wydajne niż przekazywanie ich bardzo głęboko w zagnieżdżonych wywołaniach funkcji, a taką właśnie strukturę ma program *hop-sniffer*.

Zgodnie z ustaleniami procedury pomiaru roamingu stacji klienckiej program powinien wykrywać dwa typy zdarzeń:

– Odebranie ramki typu *MGMT* i podtypu *Association Response*.

– Odebranie ramki typu *MGMT* i podtypu *Disassociation*.

Sterują one przebiegiem obserwacji zjawiska, które posiada trzy zidentyfikowane stany:

1. Stan asocjacji z początkowym punktem dostępowym.
2. Stan przełączania kanału radiowego.
3. Stan asocjacji z docelowym punktem dostępowym.

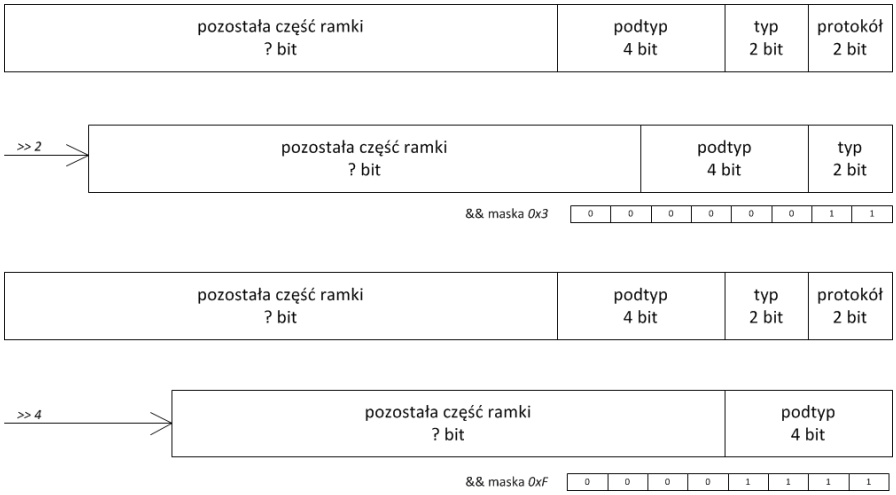
Stanom tym odpowiadają dobrze zdefiniowane zdarzenia. Odebranie ramki typu *Association Response* z adresem źródłowym MAC początkowego punktu dostępowego wprowadza scenariusz pomiarowy w stan pierwszy. Sytuacja ta zmienia się po odebraniu ramki typu *Disassociation* z adresem źródłowym MAC stacji klienckiej. Oznacza to, że stacja rozpoczęła proces roamingu i tym samym wprowadziła procedurę pomiarową w stan drugi. Ostatecznie pomiar kończy wejście scenariusza w stan trzeci spowodowane odebraniem ramki typu *Association Response* z adresem źródłowym MAC docelowego punktu dostępowego, który potwierdza asocjację przybywającej stacji klienckiej.

Pierwszą czynnością jest ekstrakcja 16-bitowego pola FC (ang. *Frame control*). Znajduje się ono na początku nagłówka, ale podczas rzutowania należy obsłużyć kodowanie *litte endian*.

Aplikacja *hop-sniffer* rozpoczyna przetwarzanie pola FC od ekstrakcji typu pakietu. W pomiarze roamingu 802.11 biorą udział jedynie ramki typu *T_MGMT*. Odczytanie typu odbywa się za pomocą makra. Wykonywane jest przesunięcie wskaźnika na pole FC poza znacznik protokołu, czyli o 2 bajty w lewo i zastosowanie do niego maski bitowej *0x3*, która odczytuje 2 pierwsze bity ramki. Jeśli odczytane bity wynoszą *0x0* (typ ramek MGMT) to program kontynuuje obsługę. Ramki nie będące typu *T_MGMT* są porzucane.

Kolejnym krokiem jest rozgałęzienie przepływu programu na obsługę wybranych podtypów ramek *MGMT*. Stosuję instrukcję warunkową *switch* z argumentem będącym odczytanym podtypem ramki. Makro ekstrakcji podtypu dokonuje przesunięcia wskaźnika na pole FC o 4 bajty w lewo (pozbywa się znacznika protokołu i typu ramki) i stosuje maskę *0xF*, która oznacza odczytanie ostatnich 4 bitów.

Ostatnią brakującą daną jest adres źródłowy MAC obsługiwanego pakietu. Najłatwiej dostać się do tej informacji poprzez rzutowanie otrzymanego w argumentach wywołania funkcji wskaźnika na początek nagłówka standardu 802.11 typu *MGMT* na jego reprezentację w postaci struktury danych języka *C* (18). Krok ten ułatwia dostęp do dwóch 6-bajtowych tablic *sa* i *da* odpowiadających źródłowemu i docelowemu adresowi MAC ramki. Pozostałe pola są zgodne ze specyfikacją i odpowiadają kolejno polu kontrolnemu, polu *duration* odpowiadającemu pozostałemu czasowi z tablicy *NAV*,



Rysunek 7: Ekstrakcja typu i podtypu z nagłówka *802.11*.

polu *BSSID* (ang. *Basic Service Set Identifier*) i numerowi kontrolnemu sekwencji.

Listing 18: Struktura *mgmt_hdr*.

```
struct mgmt_hdr
{
        u_int16_t fc;
        u_int16_t duration;
        u_int8_t da[6];
        u_int8_t sa[6];
        u_int8_t bssid[6];
        u_int16_t seq_ctrl;
};
```

Przełączanie kanału radiowego stacji pomiarowej.

Podstawową charakterystyką interfejsu karty radiowej jest jego praca wyłącznie na jednej częstotliwości w danej chwili. Popularne programy przechwytywania ramek komunikacji bezprzewodowej (np. *Wireshark*) udostępniają opcję ciągłej zmiany kanału pracy w celu obrazowania ruchu w całym spektrum dostępnym w medium transmisyjnym (ang. *channel hopping*). Z punktu widzenia aplikacji *hop-sniffer* zachowanie takie utrudniałoby i wprowadzało zakłamania kalkulacji opóźnień wybranych zjawisk. Nie zmienia to faktu, że w obserwacji przełączania częstotliwości pracy podczas roamingu 802.11 potrzebna jest jednorazowa zmiana kanału interfejsu urządzenia przechwytującego.

Jednym z możliwych rozwiązań, które wyklucza potrzebę przełączania kanału jest użycie dwóch kart radiowych pracujących odpowiednio na często-

tliwości każdego z punktów dostępowych biorących udział w eksperymencie. Ideą programu *hop-sniffer* jest jednak możliwość uruchomienia na niewielkim komputerze przenośnym i łatwa implementacja procedur pomiarowych w oparciu o pojedynczy interfejs przechwytujący.

Zastosowane rozwiązanie jest zgodne z założeniem o przeprowadzaniu scenariusza pomiarowego w odpowiedzi na wykryte zdarzenia (ramki protokołu) i wykorzystuje metodę jawnego przełączenia kanału pracy karty radiowej. Zastosowałem udostępniany przez sterownik interfejs wywołań systemowych IOCTL za pomocą, którego program nasłuchujący wystosowuje polecenie SIOCSIWFREQ służące do wprowadzenia urządzenia w podaną w jednostkach hertz częstotliwość pracy.

Procedurę można podzielić na następujące fazy:

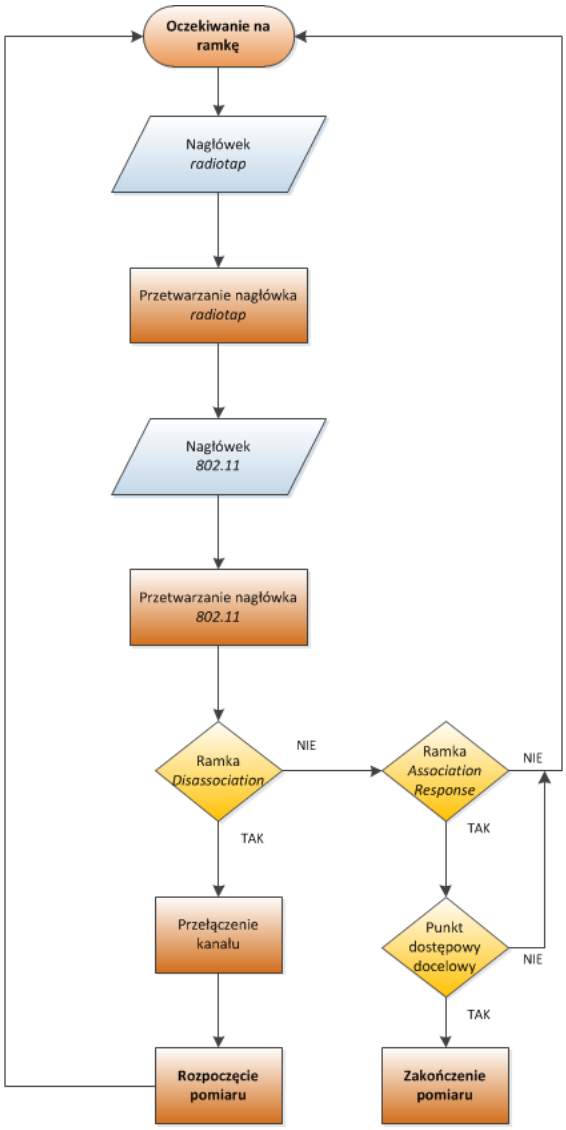
1. Otwarcie gniazda, które umożliwi wywołanie polecenia.
2. Przygotowanie argumentów wywołania w postaci akceptowanej przez polecenie.
3. Wywołanie IOCTL SIOCSIWFREQ w odpowiedzi na przetworzenie ramki typu *Disassociation*.

Program dąży do otwarcia gniazda AF_INET (ang. *Internet socket*) dostępnego na systemach Linux. Przewidziana jest też możliwość rozwoju i przyszłej przenośności aplikacji w postaci prób stworzenia innych użytecznych gniazd (IPX, AX.25, APPLETALK) w wypadku niedostępności gniazda typu *Berkeley socket*. Otwarcie następuje z parametrem SOCK_DGRAM niezbędnym do wywołania typu IOCTL.

Argumentem wywołania jest struktura *iwreq* będąca zmodyfikowaną wersją *ifreq* posiadającą dodatkowe elementy unii *u* w tym pole *freq*. Jest to pole typu *iw_freq* rozdzielające wartość częstotliwości (daną w postaci zmiennoprzecinkowej) na mantysę i wykładnik, gdyż jądro nie udostępnia arytmetyki zmiennoprzecinkowej.

Jeśli podczas przetwarzania wykryta zostanie ramka typu *Disassociation* to program *hop-sniffer* przełącza kanał interfejsu nasłuchującego na częstotliwość nowego punktu dostępowego w celu przechwycenia stempla czasowego momentu asocjacji stacji klienckiej i tym samym zakończenia roamingu 802.11.

0.6 Podsumowanie



Rysunek 8: Diagram przepływu sterowania programu *hop-sniffer*.

Bibliografia

[1] A.Barbalace, A.Lucheta, G.Manduchi, M.Moro, A.Soppelsa, and C.Taliercio. *Performance Comparison of VxWorks, Linux, RTAI and Xenomai in a Hard Real-time Application*, 2007.

[2] IEEE Standards Association. *IEEE802.11-2007: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications*, 2007.

[3] IEEE Standards Association. *IEEE802.11n-2009: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications*, 2009.

[4] Gianluca Cena, Lucia Seno, Adiano Valenzano, and Claudio Zunino. *On the Performance of IEEE 802.11e Wireless Infrastructures for Soft-Real-Time Industrial Applications*, 2010.

[5] Bert Hubert, Thomas Graf, Gregory Maxwell, Remco van Mook, Martijn van Oosterhout, Paul B Schroeder, Jasper Spaans, and Pedro Larroy. *Linux Advanced Routing & Traffic Control HOWTO*.

[6] Richard Kelly and Joseph Gasparakis. *Common Functionality in the 2.6 Linux Network Stack*, 2010.

[7] Linux manual. *die.net*.

[8] Theodore Ts'o, Darren Hart, and John Kacur. *Real-Time Linux Wiki*, 2008.

[9] Gang Wu, Sathyanarayana Singh, and Tzi cker Chiueh. *Implementation of Dynamic Channel Switching on IEEE 802.11-Based Wireless Mesh Networks*, 2008.