Politechnika Warszawska
Wydział Elektroniki I Technik Informacyjnych
Instytut Telekomunikacji

Praca dyplomowa inżynierska

Symulator Sieci WLAN

Tomasz Kabala
Dariusz Wawer

Promotor:
dr inż. Krzysztof Szczypiorski

....
ocena pracy

....
podpis Przewodniczącego Komisji

Warszawa, styczeń 2010 r.

Streszczenie

Symulator sieci WLAN.

Celem niniejszej pracy dyplomowej było zaimplementowanie symulatora sieci bezprzewodowych pozwalającego na dokładne badanie zachowania m.in. nowoczesnych protokołów sieciowych, systemów steganograficznych czy kodów nadmiarowych. Stworzony został, w języku Java (SE 1.6), symulator *EWS*. Wyniki prostych symulacji w nim przeprowadzonych zostały porównane z wynikami z symulatora *NS*. Zaimplementowane zostały, także w języku Java, dwa narzędzia pozwalające na analizę plików wynikowych symulatora *EWS*. Są to *EwsLogVisualizer*, który graficznie przedstawia proces transmisji ramek w sieci, oraz *EwsStatistic*, który pozwala na uzyskanie informacji statystycznych z pliku wynikowego.

Abstract

A WLAN network simulator.

The aim of this engineering thesis was to implement a wireless network simulator providing means to analyze behavior of modern network protocols, steganographic systems and redundancy codes. An *EWS* simulator has been programmed in Java (SE 1.6) language. Results of basic simulations conducted in *EWS* have been compared to *NS* simulator results. Two additional tools have been implemented. *EwsLogVisualizer*, which is capable graphically representing *EWS* simulation results, and *EwsStatistic*, which computes various statistical results of given simulation results.

Życiorys

Tomasz Kabala
Kierunek: Telekomunikacja
Specjalność: Systemy i sieci telekomunikacyjne
Urodzony: 22. maja 1986 r. w Otwocku.
Numer indeksu: 201551

W 2005 roku ukończyłem naukę w klasie o profilu matematyczno-fizycznym w liceum ogólnokształcącym im. K. I. Gałczyńskiego w Otwocku. W tym samym roku rozpocząłem studia na wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej na kierunku Telekomunikacja. Od 2008 roku pracuję na stanowisku administratora sieci w Instytucie Zoologii Polskiej Akademii Nauk.

…………………………….

Życiorys

Dariusz Wawer
Kierunek: Telekomunikacja
Specjalność: Systemy i sieci telekomunikacyjne
Urodzony: 1. stycznia 1986 r. w Warszawie
Numer indeksu: 200832

W 2005 roku ukończyłem XIV L.O. im. Stanisława Staszica w Warszawie. Tego samego roku rozpocząłem studia na wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej na kierunku Telekomunikacja. Od marca 2008 roku pracuję jako programista w firmie CC Otwarte Systemy Komputerowe. Byłem członkiem drużyny, która zwyciężyła Międzynarodowy Turniej Młodych Fizyków w Brisbane w Australii w 2004 roku. Jestem członkiem Koła Naukowego Twórców Gier na wydziale Elektroniki i Technik Informacyjnuch Politechniki Warszawskiej. Opublikowałem dwa artykuły o tematyce programistycznej w czasopiśmie Software Developers Journal. Główne obszary moich zainteresowań to programowanie symulacji fizycznych, gier oraz aplikacji internetowych.

....

Spis treści

Spis rysunków

Spis tabel

Rozdział 1

Wstęp

1.1 Motywacja

Istniejące symulatory sieci bezprzewodowych dają szerokie możliwości analizy ich działania, czasem nawet zapewniają narzędzia do ich projektowania. Tworzone są w różnych językach programowania, przy wykorzystaniu różnych architektur, wreszcie, są to symulatory różnego typu. Niestety, w większości przypadków dodawanie własnych funkcjonalności do tych symulatorów jest skomplikowane oraz wymaga znajomości nierzadko skomplikowanej architektury.

Koncepcja implementacji własnego symulatora narodziła się podczas dyskusji na temat symulowania systemu HICCUPS [1]. Zrobienie tego w symulatorze *NS* sprowadziłoby się jedynie do oznaczenia strumienia „ukrytego", bez uwzględniania błędnych sum CRC. Zapewne możliwa byłaby modyfikacja *NSa*, by dokładniej oddawał działanie systemu, lecz wymagałoby to zaznajomienia się ze skomplikowanym i bardzo rozbudowanym kodem. Własny symulator byłby przydatnym narzędziem, zaś, co także jest ważne, napisanie go byłoby dużym wyzwaniem.

Implementacja własnego symulatora ma dwie podstawowe zalety: pełną kontrolę nad jego architekturą, co pozwala już na etapie projektowania przystosować go do naszych wymagań.

Chcemy, by nasz symulator mógł być w przyszłości wykorzystywany do badania nowoczesnych protokołów sieciowych (na przykład tych związanych z IPv6), sieci bezprzewodowych oraz systemów steganograficznych.

Tworzony przez nas symulator nazwaliśmy *Engineer's Wireless Simulator*, w skrócie *EWS*. Jak wskazuje nazwa, staraliśmy się w inżynierski sposób podejść do problemów symulacji. Gdzie było to możliwe, dokonaliśmy uproszczeń, co widać na przykład przy założeniach modelu fizycznego. Staraliśmy się jednak by uproszczenia te nie wpływały znacząco na mechanizmy sieci bezprzewodowej.

1.2 Cel pracy

Celem naszej pracy jest zaimplementowanie symulatora sieci bezprzewodowej odzwierciedlającego rzeczywiste zależności pomiędzy poszczególnymi logicznymi elementami sieci, w szczególności między warstwami protokołów. Symulator powinien zapewnić możliwość łatwej rozbudowy: dodawania nowych protokołów, modeli fizycznych oraz modeli błędów. W symulatorze powinny być generowane oraz transmitowane prawdziwe dane, w szczególności nagłówki generowane przez protokoły sieciowe.

1.3 Układ pracy

Praca podzielona jest na kilka części. W pierwszej przedstawiony będzie proces projektowania symulatora, wraz z poczynionymi założeniami. W drugiej omówione zostaną najważniejsze elementy architektury *EWSa*. Następnie przedstawione zostaną narzędzia zaimplementowane wraz z symulatorem, które ułatwiają pracę z nim. Część czwarta dotyczyć będzie testów porównawczych symulatora *EWS* z symulatorem *NS*. Ostatni rozdział to podsumowanie pracy. Tam wskazane będą także kierunki rozwoju symulatora *EWS*.

1.4 Symulowanie sieci bezprzewodowych

Większość symulatorów sieciowych posiada moduły odpowiadające za modelowanie sieci bezprzewodowych. Poniżej postaramy się krótko scharakteryzować kilka najpopularniejszych z nich.

Network Simulator (znany także jako *ns-2*)[2] jest dyskretnym symulatorem zdarzeniowym. Topologie sieciowe opisuje się za pomocą obiektowego wariantu języka TCL (OTcl) natomiast cały symulator napisany jest w C++. Możliwe jest badanie wielu protokołów w sieciach przewodowych jak i bezprzewodowych. Symulator niestety nie jest pozbawiony błędów, które cały czas są zgłaszane do twórców i na bieżąco korygowane. Również wpływ czynników zewnętrznych nie jest dobrze odzwierciedlony co jest bardzo ważne przy symulowaniu sieci bezprzewodowych.

Kolejny znanym symulatorem jest *Modeler Wireless Suite* firmy OPNET [7]. Pozwala on na symulowania działania kilku rodzajów sieci bezprzewodowych (GSM, UMTS, 802.11, Bluetooth) oraz wielu protokołów sieciowych. Tutaj także mamy do czynienia z dyskretną symulacją zdarzeniową.

Symulatorem wyłącznie sieci bezprzewodowych jest program *Global Mobile Information System Simulator (GloMoSim)*[8]. Cała aplikacja jak i topologie symulowanych sieci napisane są w Parsecu. Jest to oparty na C język do tworzenia równoległych symulacji zdarzeniowych. Zaimplementowanych jest kilka modeli propagacji, warstw łącza danych jaki i protokołów transportowych. Osobno symulowane jest działanie każdej warstwy modelu OSI.

Rozdział 2

Projektowanie

2.1 Terminy

- log – plik wynikowy symulatora, zawierający spis wszystkich zdarzeń, które zaszły w trakcie symulacji.

- węzeł – element symulacji, posiadający miejsce w przestrzeni 2D oraz przypisany stos protokołów. Węzeł jest aktywnym oraz pasywnym uczestnikiem symulacji – może zarówno nadawać jak i odbierać transmitowane dane.

- zdarzenie – pewna czynność wykonana w symulacji, która wiąże się z utworzeniem wpisu w logach, np. wysłanie ramki lub błąd w odbiorze danych. Zdarzenia dzielą się na dwa rodzaje: fizyczne, na przykład przekłamanie pojedynczego bitu w transmisji, oraz logiczne, jak przepełnienie kolejki. Te pierwsze generowane są przez model fizyczny, te drugie przez protokoły.

- akcja – czynność wykonywana w określonej chwili czasu symulacji lub po spełnieniu pewnych warunków. Akcją może być np. upłynięcie czasu oczekiwania na ACK w warstwie MAC.

- czas symulacji – czas wewnętrzny symulatora, który upłynął od momentu rozpoczęcia symulowania.

- cykl symulacji – pojedyncza iteracja czynności aktywnych i pasywnych wszystkich węzłów symulacji.

- długość cyklu symulacji – czas symulacji przypadający na jeden cykl.

2.2 Podstawowe informacje

Podział pracy nad symulatorem był następujący:

Dariusz Wawer: projekt i implementacja architektury symylatora *EWS*, implementacja części protokołów, narzędzie *EwsLogVisualizer*, przygotowanie testów porównawczych w symulatorze *EWS*.

Tomasz Kabala: implementacja części protokołów, narzędzie *EwsStatistic*, przygotowanie testów porównawczych w symulatorach *EWS* i *NS*, składanie pracy w systemie LaTeX.

Oprócz tych zadań obaj na bieżąco testowaliśmy oraz poprawialiśmy kod symulatora. Dużo czasu poświęciliśmy także na dyskusje na temat założeń symulatora, jego architektury, działania i innych związanych z nim tematów.

W trakcie pracy stosowaliśmy następujące narzędzia:

- Eclipse – IDE programistyczne, podstawowe środowisko w którym implementowany był symulator;
- NetBeans – alternatywne IDE, wykorzystywane do tworzenia diagramów UML;
- CVS – repozytorium, w którym przechowywany jest kod źródłowy symulatora;
- Mantis – narzędzie do zarządzania projektami, tzw. *bug tracker*, wykorzystywany do planowania prac oraz zgłaszania problemów związanych z symulatorem.

2.3 Założenia algorytmu symulacji

Pierwszym krokiem było określenie założeń algorytmu symulacji, które pozwolą na wypełnienie stawianych przed symulatorem zadań. Wymienione poniżej założenia są niezmienne, architektura była projektowana pod kątek ich spełnienia.

- Symulator posiada pewną rozdzielczość, która wyrażana jest w ilości bitów na cykl symulacji. Wartość ta może być zarówno równa 1, mniejsza od 1 (kilka cykli na bit, duża dokładność, duży czas symulowania) jak i większa od 1 (jeden cykl na kilka bitów, mniejsza precyzja, krótki czas symulacji).
- Transmitowane są prawdziwe dane zero-jedynkowe. Każdy protokół musi dostarczyć prawdziwe i poprawne nagłówki do enkapsulacji w warstwie niższej. payload może być losowany.
- Sprawdzanie sum kontrolnych opiera się na obliczeniu ich na podstawie otrzymanych danych i wykonywane jest przez obiekty protokołów w odpowiednich warstwach.

2.4 Podstawowe założenia modelu fizycznego

Pierwsze implementacje klas odpowiadających za model fizyczny symulatora będą spełniały następujące założenia:

- fale propagują w dwuwymiarowym układzie współrzędnych (x,y);
- każdy nadajnik ma przypisaną moc nadawania;
- wszystkie wykorzystywane anteny są antenami izotropowymi;
- fale rozchodzą się jednakowo w każdym kierunku;
- między wszystkimi elementami sieci istnieje bezpośrednia widoczność;
- nie mają miejsca żadne odbicia;
- nie ma miejsca efekt Dopplera;
- elementy sieci są stacjonarne;
- dla symulacji zdefiniowana jest moc szumów;
- moc sygnału maleje zgodnie ze zdefiniowaną dla symulacji, dowolną funkcją odległości;
- dla symulacji jest zdefiniowana zależność BER(SNR, C), gdzie C to prędkość przesyłania danych (jedna z możliwych w 802.11g). W szczególności może istnieć takie SNR, powyżej którego BER = 0 lub taki, poniżej którego BER = 1;
- dane są transmitowane bit po bicie, sygnał z każdego z nadajników reprezentowany jest rozchodzącym się w przestrzeni pierścieniem ze środkiem w nadajniku, o szerokości odpowiadającej ilości przesyłanych danych przy danej prędkości nadawania;
- w momencie spotkania się dwóch sygnałów w odbiorniku, ten mocniejszy traktowany jest jako sygnał danych, ten słabszy jako zakłócenia, przez co dodawany jest do zdefiniowanej wcześniej mocy szumów. Wówczas znacząco zmienia się wartość SNR. W przypadku dwóch sygnałów o podobnej mocy, bezbłędny odbiór jest praktycznie niemożliwy;
- kiedy spotkają się conajmniej dwie ramki, wszystkie oprócz najsilniejszej traktowane są jako zakłócenia i ich moc jest odejmowana od mocy ramki najsilniejszej;
- nie są uwzględniane interferencje międzykanałowe.

Będzie możliwe dodanie innych implementacji, które mogą zachowywać się inaczej niż te podstawowe, czego przykładem jest zmieniony model błędów zastosowany w teście porównującym straty pakietów w zależności od prędkości transmisji.

2.5 Projektowanie symulatora

Proces projektowania symulatora był najbardziej wymagającym ze wszystkich etapów pracy nad nim. Ze względu na postawione przed *EWSem* wymagania, jego architektura musiała być jasna, spójna, logiczna i efektywna, a zarazem łatwa w rozbudowie. Przyświecały nam paradygmaty programowania obiektowego – np. tworzenie jasnej hierarchii klas, implementacja klas abstrakcyjnych, interfejsów – i programowania modularnego – wykorzystanie wymienialnych obiektów do implementacji niektórych funkcji, zamiast na stałe przypisanych metod. W trakcie tego etapu zapoznaliśmy się z literaturą związaną z programowaniem symulacji komputerowych[14][15][16]

Trudną decyzją był także wybór języka programowania. Ze względu na swoją przejrzystość oraz łatwość implementacji, wybrany został język Java. Na plus można także zaliczyć przenośność kodu, dzięki czemu symulator mógłby być uruchamiany na dowolnym systemie bez konieczności żmudnej kompilacji czy spełniania skomplikowanych zależności, co jest częste w przypadku programów napisanych w C++. Wybór Javy wiąże się jednak z konsekwencjami: niektóre operacje wykonywane w tym języku są dużo bardziej kosztowne ze względu na czas wykonania, niż np. w C czy C++, co może znacząco wydłużyć proces symulacji.

Kolejnym etapem był wybór rodzaju symulatora. Istnieje kilka kryteriów podziału symulacji komputerowych - najważniejsze z nich to podział ze ze względu na upływ czasu oraz przewidywalność zdarzeń. *EWS* jest symulatorem o czasie dyskretnym, w którym cykl symulacji (krok czasowy) jest definiowalny przez użytkownika.

Kryterium przewidywalności zdarzeń jest zależne od implementacji protokołów - mogą w nich zostać wprowadzone elementy niedeterministyczne. Przykładem takiego elementu jest algorytm *Exponential Backoff*, wykorzystywany w 802.11, który przez losowy czas czeka na ponowną próbę nadania ramki. Sam symulator jednak, w szczególności zaimplementowany model fizyczny, jest deterministyczny. Dwukrotne nadanie identycznych ramek w takich samych warunkach da te same rezultaty.

2.6 Przygotowanie symulacji

Każda symulacja jest oddzielną klasą wykorzystującą inne klasy dostępne w *EWSie*. Możliwe jest samodzielne przygotowanie wszystkich niezbędnych obiektów symulacji, lecz jest to czasochłonne. Przygotowane zostały klasy użytkowe, dzięki którym osoba chcąca zaprogramować własną symulację nie musi dobrze znać wszystkich elementów symulatora.

Podstawową klasą służącą do przeprowadzania symulacji jest *SimulationBase*. Przygotowuje ona wszystkie obiekty do tego potrzebne, dzięki czemu użytkownik musi jedynie zdefiniować węzły oraz przypisać do nich stosy protokołów. To zadanie z kolei ułatwia zestaw klas potomnych po *AbstractLayerSuite*, które przygotowują predefiniowane, typowe stosy protokołów, zaś użytkownik przypisuje im jedynie najwyższą warstwę. Przykład prostej symulacji demonstruje poniższy kod.

```
public class TwoNodesSim extends SimulationBase {

 private static final TransferRate TRANSFER_RATE = TransferRate.G1;
 private static final Channel CHANNEL = Channel.Wifi1;
 private static final int POWER = 5;

 public TwoNodesSim() {
   super((long) 3e8);
 }

 @Override
 protected void buildSimulationEvents() {
 }

 @Override
 protected void buildSimulationElements() {
   ToplessBasicPhyMacSuiteWithMacAddresses suite1 =
     new ToplessBasicPhyMacSuiteWithMacAddresses(physicalModel,
       new Node("node_1", 0, 0),
       new InterfaceSettings(CHANNEL, POWER, TRANSFER_RATE),
       new StopAndWaitSender(), 1, 2);


   ToplessBasicPhyMacSuiteWithMacAddresses suite2 =
     new ToplessBasicPhyMacSuiteWithMacAddresses(physicalModel,
       new Node("node_2",80, 0),
       new InterfaceSettings(CHANNEL, POWER, TRANSFER_RATE),
       new StopAndWaitReceiver(), 2, 1);
```

```
    suite1.connectToSimulationOverseer(so);
    suite2.connectToSimulationOverseer(so);

  }

  public static void main(String[] args) {
    new TwoNodesSim().conductSimulation();
  }
}
```

Przyjrzyjmy się po kolei zawartości tej klasy. Pierwsze trzy linijki *TwoNodesSim* to stałe określające podstawowe parametry transmisji – prędkość, kanał oraz moc nadawania. Poniżej, w konstruktorze, wywołany jest konstruktor klasy nadrzędnej, którego parametr określa długość symulacji w nanosekundach. Wartość tu podana odpowiada 300ms.

W metodzie *buildSimulationEvents* mogą być zdefiniowane zdarzenia dla symulacji. W tym przypadku metoda ta jest celowo pusta - symulacja nie uwzględnia zewnętrznych zdarzeń.

W metodzie *buildSimulationElements* definiowane są węzły oraz ich stosy protokołów. Dla każdego stosu musimy określić warstwę najwyższą, w tym wypadku protokół „Stop and Wait”, ustawienia interfejsu radiowego oraz położenie węzła. Ostatnie dwie cyfry to adresy MAC - własny, oraz adres do którego będą transmitowane ramki. Następne dwie instrukcje dołączają utworzone węzły do symulacji.

W metodzie *main* tworzymy zdefiniowany obiekt symulacji oraz wywołujemy metodę ją przeprowadzającą. Dla domyślnych ustawień zapisu logów utworzony zostanie plik *logic.log*, w którym zapisane będą wszystkie zdarzenia logiczne, oraz *phy.log*, w którym zostaną zapisane zdarzenia fizyczne. Oba pliki można otworzyć przy pomocy programów *EwsStatistic* oraz *ELV*.

Utworzona w ten sposób symulacja jest jednak w pewien sposób niepełna, gdyż, po pierwsze, nie została automatycznie przeprowadzona analiza wyników oraz, po drugie, wykonano tylko jedną symulację, a zwykle sieć badana jest względem jakiejś zmiennej, np. prędkości transmisji.

Przeprowadzenie pojedynczej symulacji połączonej z podstawową analizą jej wyników umożliwia klasa *StatRunner*.

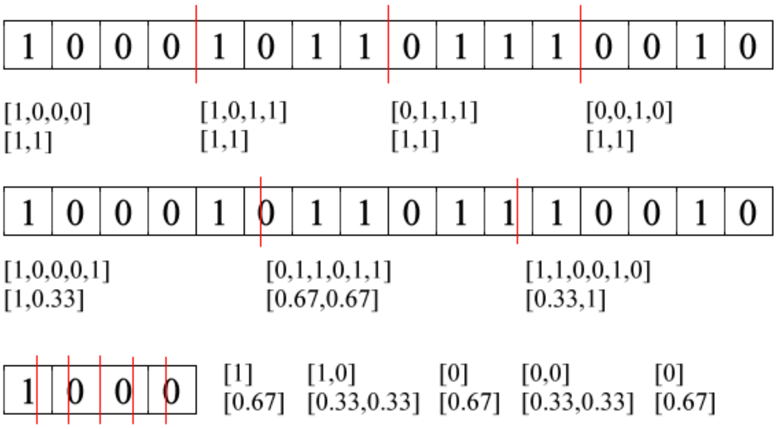
Do przeprowadzania zaawansowanych symulacji można wykorzystać klasę *MultiStatRunner*. Była ona wykorzystana także do przygotowania zestawu testów porównujących *EWS* z symulatorem *NS*, które opisane zostały w rozdziale 5. Klasa ta pozwala zdefiniować dla symulacji, oprócz węzłów i zdarzeń, m.in. metryki, które mają być zbadane czy sposób zapisu wyników. Umożliwia także automatyczne przeprowadzenie serii symulacji z uwzględnieniem kilku zmiennych.

2.7 Przebieg implementacji symulatora

Podczas pracy nad symulatorem musieliśmy sprostać kilku problemom. Na początku, tuż po zaimplementowaniu podstaw architektury, podstawowego modelu fizycznego oraz testowych protokołów okazało się, iż symulator działa niestabilnie. W szczególności algorytm łączenia przesłanych paczek danych nie działał poprawnie przy niektórych prędkościach transmisji lub długościach cyklu symulacji. Znalezienie błędu zajęło stosunkowo dużo czasu, gdyż szukaliśmy go nie tam, gdzie trzeba. Okazało się, iż sam algorytm jest poprawny – problem tkwił w operacjach na liczbach zmiennoprzecinkowych. Zmienne typu float, które były wykorzystywane do przechowywania informacji o aktualnie wysyłanym fragmencie ramki, okazały się niedostatecznie dokładne. 32 bitowa liczba zmiennoprzecinkowa zapewnia dokładne odtworzenie 7 cyfr po przecinku [3]. Czas transmisji jednego bitu przy prędkości transmisji 54 Mbps w 802.11g to ok. 1,76e-8s (17,6ns). Rozwiązaniem problemu była zmiana typu wykorzystywanych zmiennych na zmienne podwójnej precyzji double, które zapewniają dokładność 15. cyfr po przecinku.

Długo rozważanym problemem była detekcja wysłanego sygnału w tym samym cyklu przez inne węzły znajdujące się blisko. W trakcie cyklu symulacji trwającego 34ns (teoretyczny czas przesłania jednego bitu przy najwyższej prędkości osiągalnej w 802.11g) sygnał radiowy przebywa w przybliżeniu 10,6m. Załóżmy, że co najmniej dwa węzły w jednym cyklu chcą rozpocząć nadawanie. Symulator odpytuje je w określonej kolejności – jeśli pierwszy węzeł w danym cyklu rozpocznie nadawanie, co powinno dziać się z tymi, które znajdują się bliżej, niż wyliczona wyżej odległość? Czy powinny być w stanie odebrać sygnał z pierwszego węzła i wstrzymać nadawanie? Czy może nie powinny, co spowodowałoby w takim przypadku kolizję? Problem staje się jeszcze większy, gdy zwiększona zostanie długość cyklu symulacji. Może wówczas zaistnieć sytuacja, w której wszystkie węzły znajdują się w odległości na tyle małej, że mogłyby odbierać ramki wysyłane w tym samym cyklu. W tej chwili model fizyczny zachowuje się w sposób optymistyczny – zakłada, iż w takim przypadku węzły są w stanie zauważyć, że inny węzeł rozpoczął nadawanie i uniknąć kolizji. Do rozważenia w przyszłości jest utworzenie modelu, w którym czas rozpoczęcia transmisji wewnątrz cyklu byłby losowy, a co za tym idzie zmienny byłby dystans pokonany przez fale radiowe.

Stosunkowo dużo pracy kosztowało zaimplementowanie modelu wysyłanych przez węzły danych oraz implementacji algorytmu łączącego te dane z powrotem w ramki. W czasie jednego cyklu wysyłana może być bowiem różna ilość bitów, w zależności od prędkości



Rysunek 2.1: Reprezentacja danych w klasie *BitBurst*

transmisji danego węzła. Dla niskich prędkości i dużych czasów cyklu może to być ułamek bita, zaś dla dużych prędkości i małych czasów cyklu mogą to być setki, nawet tysiące bitów (symulator nie nakłada ograniczeń na czas trwania cyklu). Co za tym idzie, w czasie jednego cyklu może być kontynuowane nadawanie bitu z poprzedniego cyklu. By sprostać tym wymaganiom stworzony został następujący model danych, zawarty w klasie *BitBurst*: po pierwsze, tablica zawierająca wartości wysyłanych bitów, po drugie, dwuelementowa tablica przechowująca informację jak duża część pierwszego i ostatniego bitu została przesłana w tym cyklu. Zawartości obu tablic dla kilku przykładowych podziałów zawiera rysunek 2.1.

W każdym cyklu symulacji każdy nadający węzeł wysyła do modelu fizycznego obiekt *BitBurst*. Po zakończeniu nadawania każdy węzeł, który odebrał te dane, łączy je w ramkę. Algorytm wykonujący tą pracę zaimplementowany jest w klasie *ReceivingPacket*.

Rozdział 3

Architektura

3.1 Podstawowe informacje

Symulator *EWS* składa się z ponad 130 klas oraz interfejsów. Większość z nich to implementacje konkretnych abstrakcyjnych elementów jego architektury – przykładowo interfejs *PhysicalModel* jest implementowany przez 6 klas, z których każda ma inne działanie.

Równolegle z symulatorem tworzone były projekty *ELV* (*EWSLogVisualizer*) oraz *EWSStatistic*, które służą do analizy plików wynikowych symulatora. Więcej informacji o tych narzędziach znajduje się w rozdziale 4.

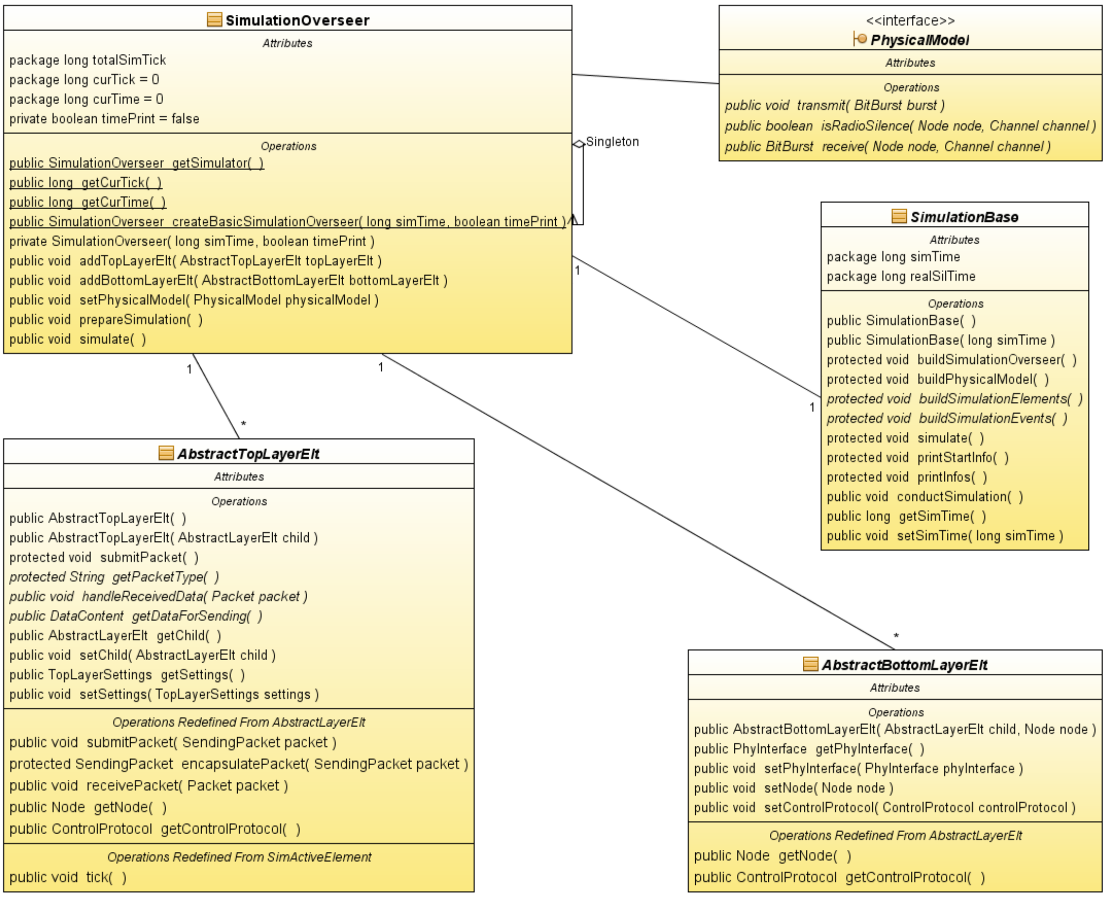
Klasy abstrakcyjne mają na celu zminimalizowanie pracy oraz wiedzy o symulatorze niezbędnej do implementacji konkretnego rozwiązania. Przykładowo, by dodać do symulatora nowy protokół, należy utworzyć nową klasę rozszerzającą *AbstractMiddleLayerElt* oraz zaimplementować w niej dwie metody narzucone przez klasę nadrzędną. Logika, która będzie się za nimi kryła, może być bardzo skomplikowana – lecz osoba ją programująca nie musi znać szczegółów działania samego symulatora.

Obiekty, z których składa się symulator, dzielą się na kilka kategorii. Podany zostanie krótki opis każdej z nich, jej funkcja oraz kilka przykładowych klas czy interfejsów do niej przynależących.

- obiekty nadzorcze – klasy odpowiedzialne za przebieg symulacji lub niezbędne do jej przeprowadzenia – *SimulationOverseer*, *ActionManager*, *EWSEventLogger*. Klasy te realizują wzorzec singletonu.

- elementy modelu fizycznego – interfejsy *BerFunction*, *SnrFunction*, *PhysicalModel* oraz ich implementacje.

- reprezentacje warstw sieci – klasy abstrakcyjne *AbstractLayerElt*, *AbstractBottomLayerElt*, *AbstractMiddleLayerElt*, *AbstractTopLayerElt*, *AbstractInterface*, *Control-*



Rysunek 3.1: *SimulationOverseer* - diagram UML

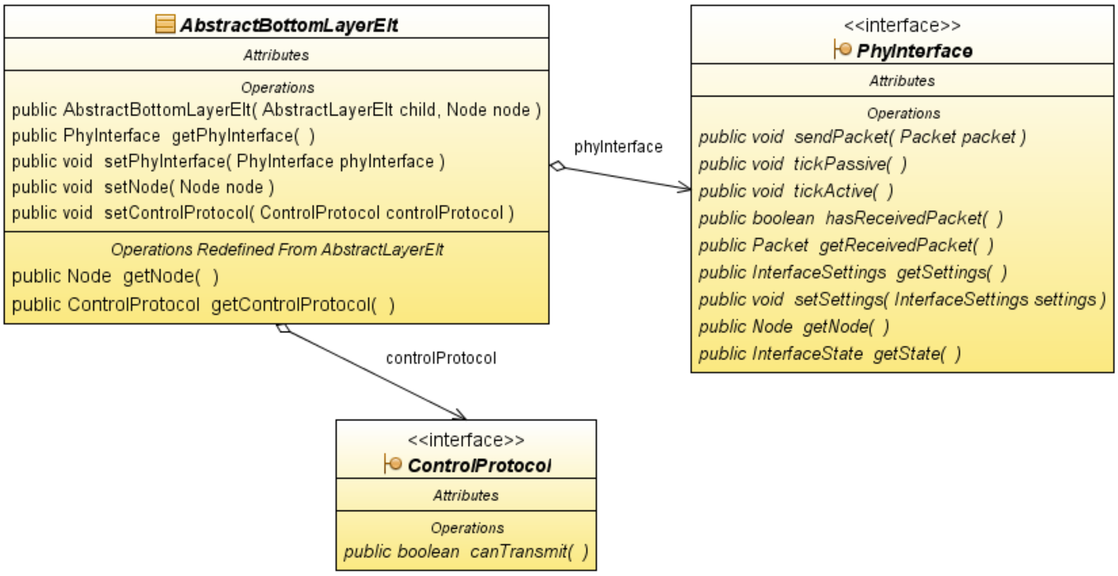
Protocol.

- reprezentacje danych – *Packet* (implementacje: *PacketImpl*, *SendingPacket*), *BitBurst*, *DataContent*, *ReceivingPacket*, *Node*. Zawierają określone informacje, które są przekazywane z jednego obiektu do drugiego (np. *Packet* przekazywany do kolejnych warstw sieci).

- zdarzenia – klasy rozszerzające *EwsEvent*, odpowiadają zdarzeniom w symulacji np. *PacketSentEvent*, *PacketReceivedEvent*, etc.

- użytkowe – *BaseSim*, *AbstractPacketBuilder*, *PacketReceiver*, *TableData*, *IntUtils*, *CRC16* – pozwalają na sprawne i efektywne wykonywanie pewnych czynności.

Główną klasą symulatora jest *SimulationOverseer*, który odpowiada za jej przebieg. Zawiera referencje do wszystkich węzłów w symulacji, wykonuje wszystkie operacje związane z jej przebiegiem. Schemat UML widoczny jest na ilustracji 3.1.



Rysunek 3.2: *BottomLayer* - diagram UML

SimulationBase jest abstrakcyjną klasą użytkową ułatwiającą przygotowywanie symulacji. Tworzy wszystkie niezbędne do jej przeprowadzenia obiekty oraz je konfiguruje. Wymaga od klasy potomnej zdefiniowania jedynie metod przygotowujących węzły symulacji oraz jej akcje.

PhysicalModel jest odpowiedzialny za transmisję danych w symulatorze. Szczegóły jego działania są zależne od implementacji. Jego metody *transmit* oraz *receive* są wywoływane przez obiekty *PhyInterface*, zaś *isRadioSilence* przez *ControlProtocol*. Zależności między *AbstractBottomLayerElt*, *PhyInterface* oraz *ControlProtocol* widoczne są na ilustracji 3.2.

AbstractTopLayerElt i *AbstractBottomLayerElt* reprezentują odpowiednio najwyższą oraz najniższą warstwę stosu protokołów. Pomiędzy nimi może być dowolna ilość warstw pośrednich *AbstractMiddleLayerElt*, lecz żadna z nich nie będzie aktywnie wywoływana przez symulator.

3.2 Przebieg symulacji

Symulacja dzieli się na trzy etapy: przygotowanie, kiedy tworzone są wszystkie niezbędne obiekty, symulacja właściwa oraz ewentualna analiza wyników przy pomocy metryk z *EwsStatistic*.

Na przygotowanie składa się:

- zdefiniowanie nadzorcy symulacji (*SimulationOverseer*);
- zdefiniowanie modelu fizycznego wraz z modelem błędów i szumów (*PhysicalModel*,

BerFunction, *SnrFunction*);

- utworzenie węzłów biorących udział w symulacji;
- zdefiniowanie zdarzeń w symulacji.

Podczas symulacji czynności wykonywane są w następującej kolejności:

- zdarzenia, które przypadają na dany cykl symulacji;
- wywołanie najwyższej warstwy, w celu sprawdzenia, czy wygenerowany został w danym cyklu pakiet;
- wywołanie najniższej warstwy, w celu wysłania danych;
- wywołanie najniższej warstwy, w celu odbioru danych;
- przesunięcie aktualnego czasu symulacji.

Po upłynięciu określonego czasu wewnętrznego symulacji symulator przechodzi do czynności końcowych. Zapisuje niezapisane jeszcze zdarzenia, zamyka pliki, itd. W zależności od symulacji może także dokonać analizy logów oraz zapisać jej wyniki.

3.3 Opisy ważniejszych klas

Poniższe klasy zawierają najważniejsze dla logiki symulatora elementy:

- *EwsSettings* – zawiera podstawowe ustawienia symulatora, na przykład: rozdzielczość, poziom szumów tła, czy prędkość propagacji fal w przestrzeni.
- *EwsLogger* – klasa użytkowa, wypisuje informacje dotyczące działania symulatora. Używana do znajdowania błędów.
- *EwsEventLogger* – zapisuje zdarzenia wygenerowane w trakcie symulacji.
- *ActionManager* – wykonuje akcje zdefiniowane w symulacji lub utworzone w trakcie jej trwania.
- *AbstractLayerSuite* – zapewnia API przydatne do konstruowania stosu protokołów. Pozwala wygodnie je łączyć oraz dodawać wynikowy stos do symulacji.

- *PacketBuilder* (oraz *PacketReceiver*) – API do generowania pakietów oraz ramek. Zapewnia możliwość zdefiniowania pól w pakiecie/ramce, przypisania im konkretnych wartości oraz wygenerowania na ich podstawie zgodnego z definicją pakietu czy ramki. *PacketReceiver* pozwala pobrać z odebranego pakietu konkretne pola na podstawie informacji zawartych w odpowiadającym mu obiekcie *PacketBuilder*.
- *ChanneledAdvPhysicalModel* – model fizyczny spełniający założenia symulatora.
- *SimulationBase* – klasa pozwalająca na łatwe przygotowanie scenariusza symulacji.
- *ReceivingPacket* – klasa służąca do gromadzenia oraz łączenia odebranych fragmentów ramek.

3.4 Ustawienia symulatora

Zmiana ustawień symulatora odbywa się poprzez nadanie odpowiednim zmiennym wartości. Nie jest to niezbędne, gdyż każda z nich ma przypisane pewne domyślne wartości. Poniżej przedstawiona jest lista klas w których znajdują się najważniejsze ustawienia symulacji oraz zmiennych w nich zawartych.

EwsSettings zawiera podstawowe ustawienia symulatora:

- *resPower* – wyznacza rozdzielczość symulatora. W każdym cyklu symulacji transmitowane jest, przy najwyższej prędkości zdefiniowanej w *TransferRate*, 2^{resPower} bitów;
- *limitedQueueSize* – określa domyślny rozmiar kolejki ramek/pakietów oczekujących na wysłanie;
- *fpPrecision* – definiuje deltę dla operacji zmiennoprzecinkowych. Wykorzystywana przy transmitowaniu ramki do sprawdzania, czy całość ramki została już wysłana. Wartość ta nie powinna być modyfikowana;
- *noiseLevel*[dBm] – poziom szumów tła. Wykorzystywany do obliczania SNR odbieranych transmisji;
- *wavePropagationSpeed*[m/s] – prędkość rozchodzenia się fal radiowych w przestrzeni. Wykorzystywana przez modele fizyczne;
- *burstLife*[m] - definiuje odległość, po przebyciu której wysyłane dane są usuwane z modelu fizycznego. Powinna być zawsze większa od maksymalnego zasięgu systemu który jest testowany;

- *timeUnitsPerSecond* – określa ilość jednostek czasu symulatora przypadających na jedną sekundę. W przypadku badania systemów o niskich prędkościach transmisji wartość ta może być obniżona. Wartością domyślną jest 10^9, co odpowiada jednej nanosekundzie.

EWSEventLogger zawiera ustawienia dotyczące plików wynikowych symulatora oraz rodzaju zapisywanych zdarzeń:

- *phyLevelFilename* – nazwa pliku w którym mają być zapisywane zdarzenia fizyczne;
- *logicLevelFilename* – nazwa pliku w którym mają być zapisywane zdarzenia logiczne;
- *savePhy* – czy zdarzenia fizyczne mają być zapisywane;
- *saveLog* – czy zdarzenia logiczne mają być zapisywane.

TransferRate - typ wyliczeniowy, w którym definiowane są obiekty odpowiadające różnym rodzajom transmisji. Określana jest ich prędkość. Poprzez sprawdzenie rodzaju transmisji możliwe jest rozróżnienie jego działania przez model fizyczny. Mimo, iż prędkości nadawania niektórych trybów z 802.11b i 802.11g są takie same, nie oznacza to, że oba te systemy muszą być w ten sam sposób traktowane przez model fizyczny.

Rozdział 4

Narzędzia pomocnicze

4.1 EwsLogVisualizer

Jest narzędziem służącym do analizy logów symulatora, w szczególności do analizy przebiegu symulacji krok po kroku. Pozwala na wizualizację przebiegu symulacji. Węzły widoczne są jako punkty, zaś transmitowane dane przedstawiane są jako rozchodzące się w przestrzeni 2D pierścienie.

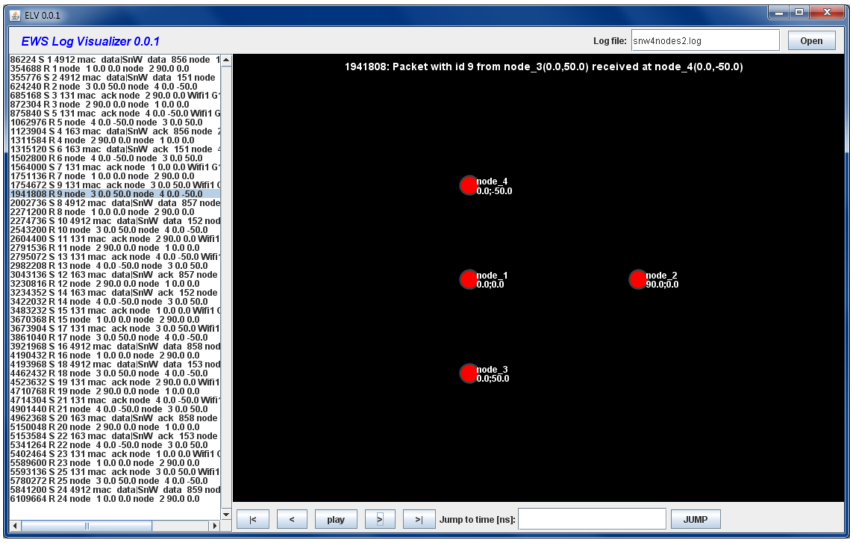
Program ten został napisany, podobnie jak symulator, w języku Java. Wykorzystuje niektóre klasy z *EWS*, w szczególności te odpowiadające za zapisywanie i wczytywanie zdarzeń z logów.

By rozpocząć korzystanie z programu należy kliknąć przycisk *open* w prawym górnym rogu aplikacji i wybrać plik z logami z którym będzie się pracowało. Po lewej stronie ekranu pojawi się lista wpisów z logów, zaś na środku zostanie wyświetlony węzeł którego dotyczy pierwszy wpis w logach. Przyciski na dole ekranu pozwalają przechodzić do poprzedniego/następnego wpisu w logach. Pole tekstowe obok nich pozwala na przejście do dowolnego czasu w logach. Klikając na wpis w liście po lewej stronie można przejść do dowolnego wpisu w logach.

Program *ELV* został zaimplementowany by umożliwić wizualizację przebiegu symulacji. Może być wykorzystywany w dwóch podstawowych celach: edukacyjnym, by dowiedzieć się jak działają pewne protokoły lub sieć wifi, oraz w celu odnalezienia błędów w implementacji protokołów. Podczas prac nad symulatorem *ELV* był bardzo pomocny m.in. podczas badania poprawności zachowania mechanizmu RTS/CTS w 802.11 [4].

ELV jest, pod względem programistycznym, typowym narzędziem użytkowym napisanym w języku Java. Jego GUI wykorzystuje graficzne komponenty z biblioteki Swing, które są w tej chwili standardem dla Javy.

W jego kodzie ciekawe są jednak dwa elementy. Pierwszym z nich jest klasa *LogFi-*



Log file: snw4nodes2.log

```
86224 S 1 4912 mac  data|SnW  data  856 node  1
354688 R 1 node  1 0.0 0.0 node  2 90.0 0.0
355776 S 2 4912 mac  data|SnW  data  151 node
624240 R 2 node  3 0.0 50.0 node  4 0.0 -50.0
685168 S 3 131 mac  ack node  2 90.0 0.0 Wifi1 G
872304 R 3 node  2 90.0 0.0 node  1 0.0 0.0
875840 S 5 131 mac  ack node  4 0.0 -50.0 Wifi1 G
1062976 R 5 node  4 0.0 -50.0 node  3 0.0 50.0
1123904 S 4 163 mac  data|SnW  ack  856 node
1311584 R 4 node  2 90.0 0.0 node  1 0.0 0.0
1315120 S 6 163 mac  data|SnW  ack  151 node
1502800 R 6 node  4 0.0 -50.0 node  3 0.0 50.0
1564000 S 7 131 mac  ack node  1 0.0 0.0 Wifi1 G
1751136 R 7 node  1 0.0 0.0 node  2 90.0 0.0
1754672 S 9 131 mac  ack node  3 0.0 50.0 Wifi1
1941808 R 9 node  3 0.0 50.0 node  4 0.0 -50.0
2002736 S 8 4912 mac  data|SnW  data  857 node
2271200 R 8 node  1 0.0 0.0 node  2 90.0 0.0
2274736 S 10 4912 mac  data|SnW  data  152 nod
2543200 R 10 node  3 0.0 50.0 node  4 0.0 -50.0
2604400 S 11 131 mac  ack node  2 90.0 0.0 Wifi1
2791536 R 11 node  2 90.0 0.0 node  1 0.0 0.0
2795072 S 13 131 mac  ack node  4 0.0 -50.0 Wifi
2982208 R 13 node  4 0.0 -50.0 node  3 0.0 50.0
3043136 S 12 163 mac  data|SnW  ack  857 node
3230816 R 12 node  2 90.0 0.0 node  1 0.0 0.0
3234352 S 14 163 mac  data|SnW  ack  152 node
3422032 R 14 node  4 0.0 -50.0 node  3 0.0 50.0
3483232 S 15 131 mac  ack node  1 0.0 0.0 Wifi1
3670368 R 15 node  1 0.0 0.0 node  2 90.0 0.0
3673904 S 17 131 mac  ack node  3 0.0 50.0 Wifi1
3861040 R 17 node  3 0.0 50.0 node  4 0.0 -50.0
3921968 S 16 4912 mac  data|SnW  data  858 nod
4190432 R 16 node  1 0.0 0.0 node  2 90.0 0.0
4193968 S 18 4912 mac  data|SnW  data  153 nod
4462432 R 18 node  3 0.0 50.0 node  4 0.0 -50.0
4523632 S 19 131 mac  ack node  2 90.0 0.0 Wifi1
4710768 R 19 node  2 90.0 0.0 node  1 0.0 0.0
4714304 S 21 131 mac  ack node  4 0.0 -50.0 Wifi
4901440 R 21 node  4 0.0 -50.0 node  3 0.0 50.0
4962368 S 20 163 mac  data|SnW  ack  858 node
5150048 R 20 node  2 90.0 0.0 node  1 0.0 0.0
5153584 S 22 163 mac  data|SnW  ack  153 node
5341264 R 22 node  4 0.0 -50.0 node  3 0.0 50.0
5402464 S 23 131 mac  ack node  1 0.0 0.0 Wifi1
5589600 R 23 node  1 0.0 0.0 node  2 90.0 0.0
5593136 S 25 131 mac  ack node  3 0.0 50.0 Wifi1
5780272 R 25 node  3 0.0 50.0 node  4 0.0 -50.0
5841200 S 24 4912 mac  data|SnW  data  859 nod
6109664 R 24 node  1 0.0 0.0 node  2 90.0 0.0
```

1941808: Packet with id 9 from node_3(0.0,50.0) received at node_4(0.0,-50.0)

Rysunek 4.1: Widok interfejsu narzędzia *ELV*

leReader, która w inteligentny sposób pre-parsuje plik z logami pozwalając na szybszy dostęp do danego zdarzenia, oraz implementuje zaawansowany mechanizm cache'owania, by zminimalizować wykorzystanie dysku podczas pracy z nim.

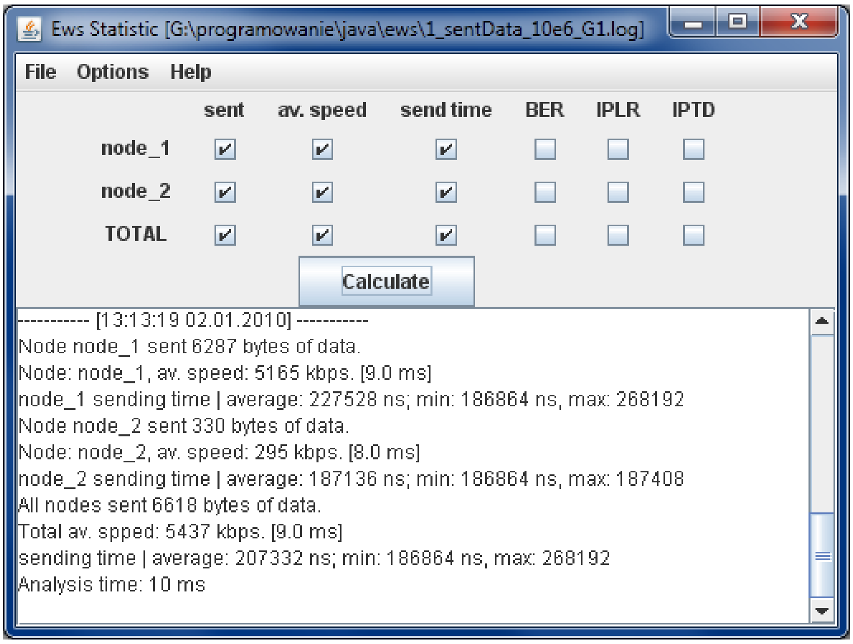
Drugi, to wykorzystanie elementów autorskiego frameworku *Scythe2DEngine*. Został on stworzony z myślą o tworzeniu i wizualizowaniu symulacji fizycznych oraz gier, ale jego część graficzna ma zastosowanie także tutaj.

4.2 EwsStatistic

Jest narzędziem dającym możliwość analizy statystycznej logów. Także został napisany w języku Java. W wersji podstawowej umożliwia wydobycie z pliku wynikowego informacji takich jak ilość danych wysłanych przez węzeł, metryki ILPR czy IPTD.

Obsługa programu sprowadza się do wskazania lokalizacji pliku z interesującymi nas logami. Wybrany plik jest analizowany pod względem występujących w nim węzłów a następnie wyświetlany jest panel z listą metryk do wyznaczenia. Możliwe jest wykonanie obliczeń dla pojedynczych węzłów jak i dla całej sieci.

Architektura aplikacji pozwala na bardzo proste rozszerzanie jej o kolejne metryki. Niezależnie od tego ile jest dostępnych metryk ani ile z nich wybierze użytkownik, plik z logami analizowany jest tylko dwa razy, co znacząco wpływa na wydajność aplikacji. Pierwsze przeszukanie ma za zadanie wykryć wszystkie węzły, które przynajmniej raz są aktywne podczas całej symulacji. W drugim przebiegu każdy wpis jest przekazywany do obiektów reprezentujących wybrane przez użytkownika metryki. Obiekty te same decydują o tym jakie dane są dla nich istotne i w jaki sposób należy jest uwzględnić w obliczeniach.



	sent	av. speed	send time	BER	IPLR	IPTD
node_1	☑	☑	☑	☐	☐	☐
node_2	☑	☑	☑	☐	☐	☐
TOTAL	☑	☑	☑	☐	☐	☐

```
----------- [13:13:19 02.01.2010] -----------
Node node_1 sent 6287 bytes of data.
Node: node_1, av. speed: 5165 kbps. [9.0 ms]
node_1 sending time | average: 227528 ns; min: 186864 ns, max: 268192
Node node_2 sent 330 bytes of data.
Node: node_2, av. speed: 295 kbps. [8.0 ms]
node_2 sending time | average: 187136 ns; min: 186864 ns, max: 187408
All nodes sent 6618 bytes of data.
Total av. spped: 5437 kbps. [9.0 ms]
sending time | average: 207332 ns; min: 186864 ns, max: 268192
Analysis time: 10 ms
```

Rysunek 4.2: Widok interfejsu narzędzia *EwsStatistic*

EwsStatistic był bardzo pomocny podczas implementacji symulatora, ponieważ pozwalał nam wstępnie ocenić czy wyniki naszych symulacji zgadzają się z tym, czego oczekiwaliśmy. Zaimplementowanie części metryk było punktem wyjścia przy tworzeniu pakietu testów *NsCompareTests*, dzięki któremu mogliśmy porównać działanie *EWSa* z symulatorem *NS*.

Rozdział 5

Testy

Chcąc wykorzystywać *EWS* w praktyce, konieczne było jego przetestowanie. Najprostszym sposobem aby to osiągnąć było porównanie wyników jego działania z wynikami innego symulatora. Szukając odpowiedniego wzorca nasz wybór padł na *ns-2* [2], jako że jest to jeden z popularniejszych symulatorów. Spośród możliwych scenariuszy testowych wybraliśmy te, które badają najważniejsze parametry działania sieci bezprzewodowej i pozwolą nam zweryfikować podstawowe mechanizmy działania symulatora *EWS*.

5.1 Scenariusze testowe

Do przeprowadzeniach testów wykorzystaliśmy topologię składająca się z dwóch węzłów sieci bezprzewodowej. Zastosowaliśmy dwa warianty rozmieszczenia węzłów:

- mały dystans (10 m) – dzięki czemu podczas transmisji nie występują błędy i możemy skupić się na badaniu właściwości protokołów;
- duży dystans (80 m) – zaczynają pojawiać się straty wynikające ze znacznej odległości pomiędzy węzłami, co pozwoli na badanie modelu błędów.

Szczegółowe parametry symulacji zostaną opisane w dalszej części rozdziału.

5.2 Mierzone metryki

W ramach testów chcieliśmy zbadać i porównać z wynikami symulatora *NS* podstawowe cechy sieci wifi:

- czas transmisji ramki – jest to najbardziej podstawowa wartość, gdyż weryfikuje poprawność mechanizmu transmisji danych;

- osiąganą przepustowość – łączy ze sobą mechanizmy transmisji oraz badanie protokołów sieciowych;
- stopa błędów – pozwala porównać model błędów zastosowany w *EWS* z tym zastosowanym w *NS*.

5.3 Wykorzystane narzędzia

Na potrzeby tej części pracy przygotowaliśmy w symulatorze *EWS* pakiet testów *NsCompareTests*. Każdy test uruchomia wielokrotnie odpowiednią symulację, za każdym razem zmieniając jej parametry, a wyniki zapisuje w pliku tekstowym. Wyniki symulacji wyliczane są za pomocą odpowiednich klas programu *EwsStatistic*.

Analogiczne symulacje stworzyliśmy także dla symulatora *NS*. Do wyliczania metryk wykorzystaliśmy odpowiedni skrypt napisany w języku Perl, który uruchamia symulacje i analizuje stworzone przez nią logi. Gotowe wyniki zapisywane są w pliku tekstowym. Przy tworzeniu symulacji kierowaliśmy się wskazówkami z [6].

Do stworzenia wykresów ilustrujących wyniki naszych badań wykorzystaliśmy program *gnuplot* [5]. W przypadku obu symulatorów już w trakcie wyliczania metryk zapisywaliśmy wyniki w formacie odpowiednim dla tego programu.

5.4 Parametry symulacji

W przypadku badania dwóch pierwszych metryk wybraliśmy wariant topologii z małym dystansem pomiędzy węzłami, by ew. błędy transmisji nie zaniżały otrzymanych wyników. W symulatorze *NS* korzystaliśmy z modelu propagacji w wolnej przestrzeni *FreeSpace*.

Przy pomiarze stopy błędów zwiększyliśmy dystans aby straty ramek były znaczące. W symulatorze *NS* aby uzyskać losową charakterystykę strat wykorzystaliśmy model propagacji *Shadowing*, w którym zasięg nadajnika nie jest już idealnym okręgiem jak to jest w przypadku *FreeSpace*. Dodatkowo dla *EWS* badaliśmy oba modele błędów jakie posiada: *BasicBerFunction* (symuluje straty bitowej stopy błędów zależnej od mocy odbieranej) oraz *TwoStateBerFunction* (model wzorowany na *NS*, gdzie zakłócenia mają charakter okresowy).

We wszystkich przypadkach testowych transmisja zachodziła tylko w jedną stronę – od nadajnika do odbiornika (nie licząc oczywiście odpowiednich potwierdzeń, charakterystycznych dla zastosowanych protokołów). Obiektem naszych badań był właśnie ten ruch. Payload pakietów wynosił zawsze 512 bajtów, co było oczywiście powiększane o odpowiednie nagłówki warstw sieci wykorzystanych w symulacji.

Prędkość transmisji [Mbps]	1, 2, 5.5, 6, 9, 11, 12, 18, 24, 36, 48, 54
Czas symulacji [s]	10
Odstęp pomiędzy pakietami [s]	0.02

Tabela 5.1: Parametry dla scenariusza *czas transmisji ramki*

Prędkość transmisji [Mbps]	1, 9, 24, 54
Czas symulacji [s]	0.01, 0.03, 0.06, 0.1, 0.3, 0.6, 1, 3, 6, 10

Tabela 5.2: Parametry dla scenariusza *przepustowość*

W celu zbadania czasu transmisji ramki i stopy błędów potrzebowaliśmy prostego źródła, które, co pewien stały okres czasu, będzie wysyłać pojedynczy pakiet. W symulatorze *EWS* to zadanie spełnia klasa *PeriodicSender*, natomiast w *NS* aplikacja CBR. Odstęp pomiędzy pakietami wynosił 0,02 s.

Aby zmierzyć maksymalną przepustowość wymagane było użycie źródła, które będzie wykorzystywało całą dostępną przepustowość. W *EWS* do tego zadania odpowiednia wydawała się klasa *StopAndWaitSender* symulująca działanie protokołu dostępowego „Stop and Wait”, w *NS* użyliśmy aplikacji FTP w połączeniu z protokołem transportowym TCP. Szerokość okna TCP ustawiliśmy na jeden, aby uzyskać takie samo działanie jak w „Stop and Wait”.

Wszystkie źródła pakietów działały w parze z odpowiednimi dla nich odbiornikami. We wszystkich transmisjach wykorzystywana była długa preambuła.

Tabele 5.1 - 5.4 zawierają szczegółowe parametry wszystkich symulacji.

5.5 Wyniki

Wyniki pierwszego testu, widoczne na wykresie 5.1, odbiegają od siebie. Szczególnie w przypadku ramek wysyłanych z dużą prędkością różnice są bardzo znaczące - dla prędkości 56 Mbps czas transmisji ramki w symulatorze *EWS* jest ponad dwukrotnie krótszy niż w symulatorze *NS*. Różnice te nas niepokoiły, więc obliczyliśmy ile powinien wynosić czas transmisji ramki o określonej długości, Czas transmisji preambuły to 0,192 ms, zaś

Prędkość transmisji [Mbps]	1, 2, 5.5, 6, 9, 11, 12, 18, 24, 36, 48, 54
Czas symulacji [s]	10
Moc nadawania [dBm]	-6, -3, 0, 7, 10, 13, 15, 17, 20

Tabela 5.3: Parametry dla scenariusza *ramkowa stop błędów*

Rozmiar pakietu [bajty]	512
Mechanizm RTS/CTS	wyłączony
Preambuła	krótka

Tabela 5.4: Parametry wspólne dla wszystkich scenariuszy

Rate [Mbps]	EWS [ms]	NS [ms]	Matematyka [ms]
54	0,26	0,67	0,27
48	0,27	0,68	0,28
36	0,30	0,71	0,31
24	0,35	0,77	0,36
18	0,41	0,83	0,42
12	0,52	0,95	0,54
11	0,55	0,98	0,57
9	0,63	1,07	0,65
6	0,85	1,31	0,88
5,5	0,91	1,38	0,94
2	2,18	2,77	2,26
1	4,17	4,95	4,33

Tabela 5.5: Czas transmisji pojedynczej ramki

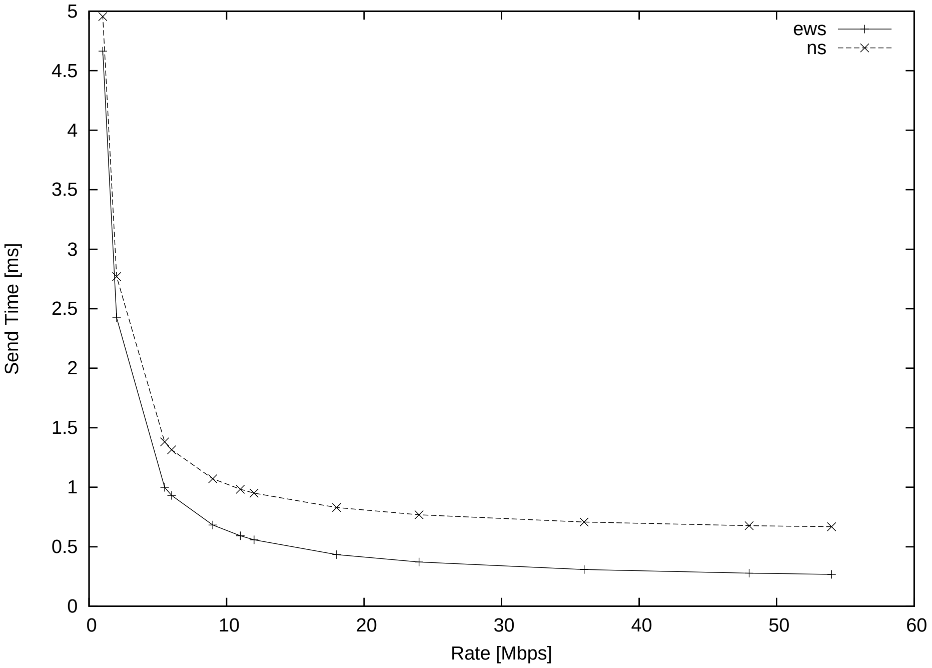
na jeden bit przy 56mbps przypada ok 17,6 ns. Wyniki obliczeń widoczne są w tabeli 5.5

Widzimy na nich, iż czasy transmisji w symulatorze *EWS* niemalże zgadzają się z wyliczonymi czasami transmisji. Nie wiemy czemu wyniki z *NSa* tak znacząco różnią się od przewidywanych.

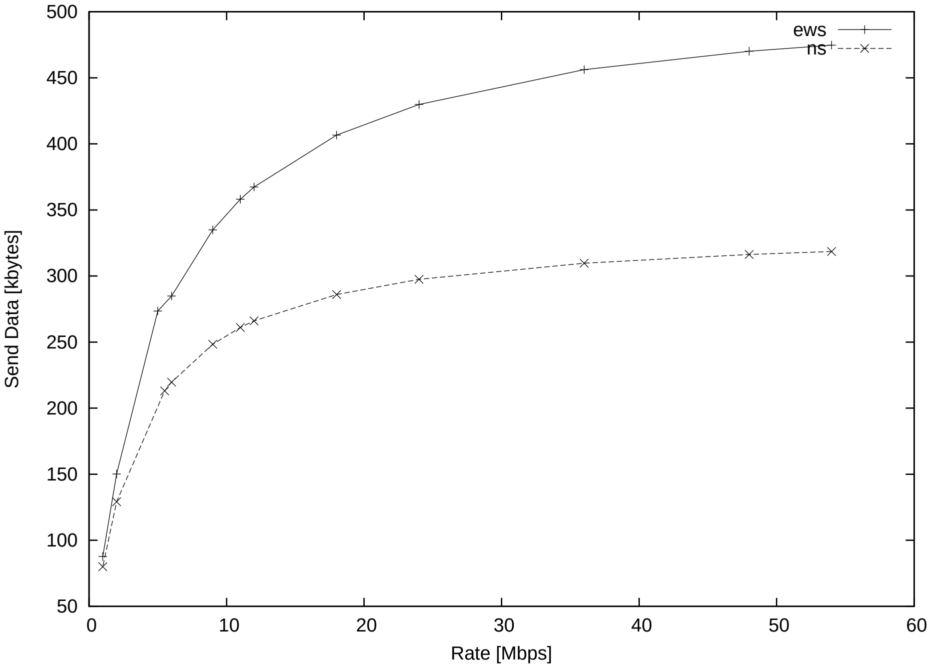
Na wykresie 5.2 można zauważyć, że charakterystyka ilości przesłanych danych w ciągu jednej sekundy w zależności od prędkości transmisji ma taki sam kształt dla obu symulatorów. *Ews* daje większe wyniki, co może być spowodowane właśnie różnicą czasów nadawania ramek, którą zauważyliśmy w scenariuszu 1.

Dodatkowo wykonaliśmy szereg testów dla krótkich jak i długich czasów symulacji. Jak się można było spodziewać, ilość przesłanych danych rośnie wtedy liniowo wraz z czasem (wykres 5.3).

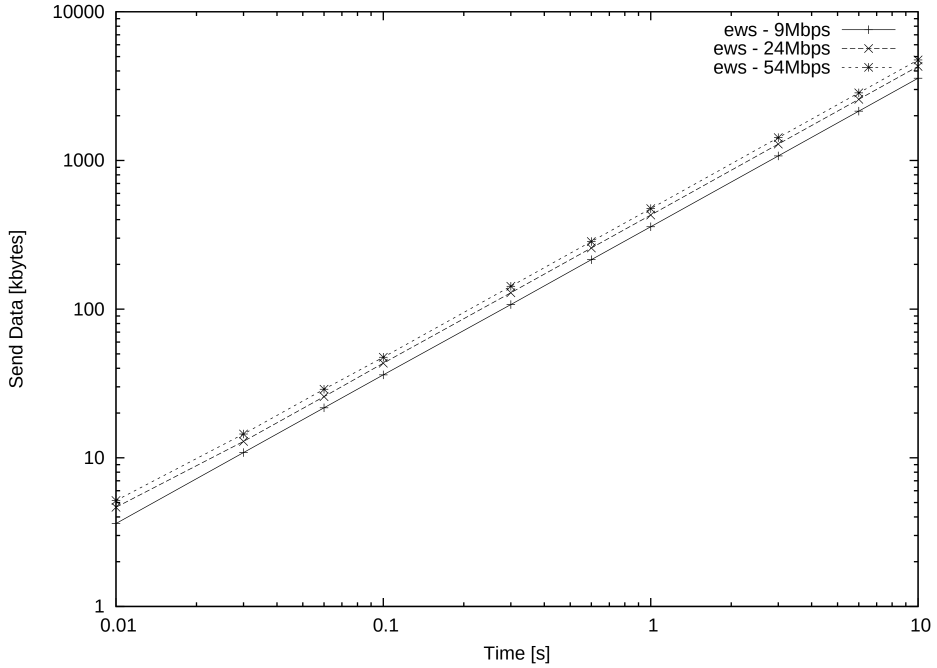
Podczas wykonywania kolejnego testu zauważyliśmy, że model błędów *BasicBerFunction* zastosowany w *EWS* zachowuję zupełnie inaczej niż niż ten wykorzystywany w *NS*. Model *BasicBerFunction* powoduje przekłamanie losowych bitów (prawdopodobieństwo zależne od wartości SNR), co powoduje większe straty przy dużych prędkościach (wykres 5.5). *NS* natomiast więcej błędów odnotowuje przy niskich prędkościach. Podobnym dzia-



Rysunek 5.1: Średni czas transmisji ramki w zależności od prędkości



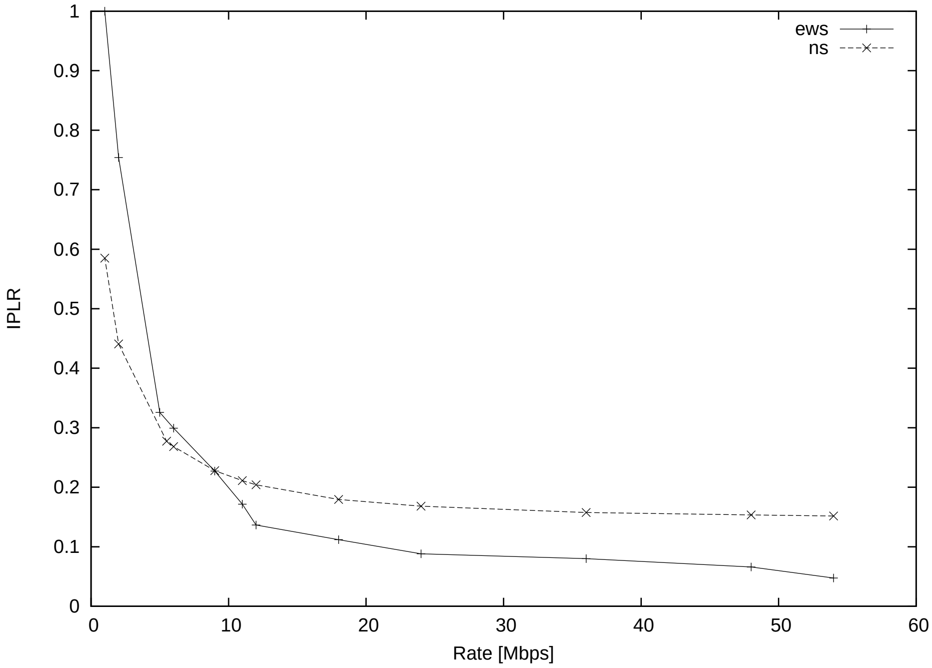
Rysunek 5.2: Ilość przesłanych danych w czasie t=1s



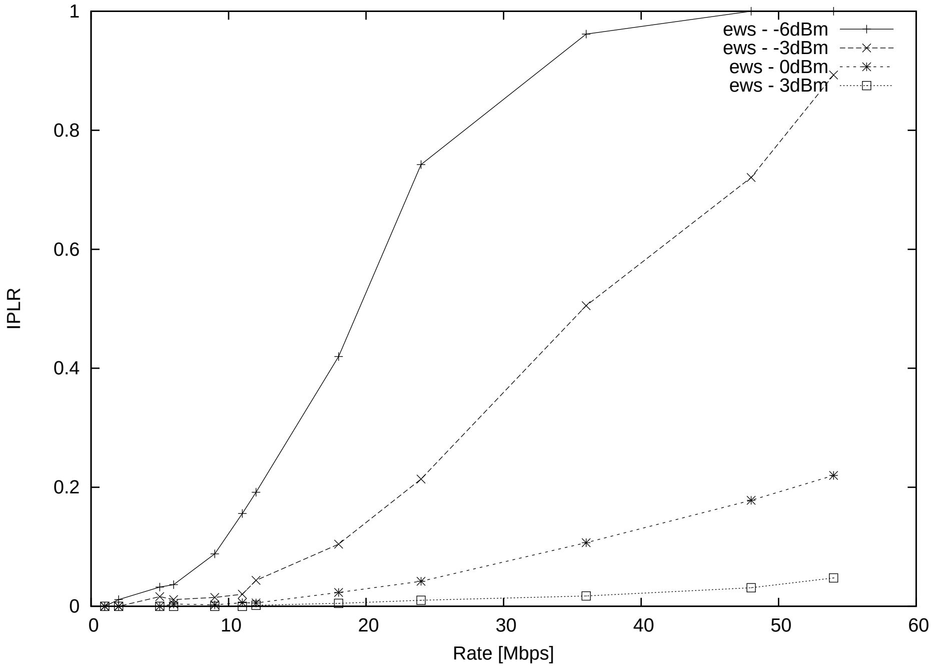
Rysunek 5.3: Ilość przesłanych danych dla wybranych prędkości transmisji w *EWS*

łaniem charakteryzuje się inny model błędów wykorzystywany przez *EWS* czyli *TwoStateBerFunction*. Jest to dwustanowy model błędów, w którym określane są dwa przedziały czasu: w którym bity są przekłamywane, oraz w którym są poprawnie odbierane. Wyniki jego zastosowania widać na wykresie 5.4.

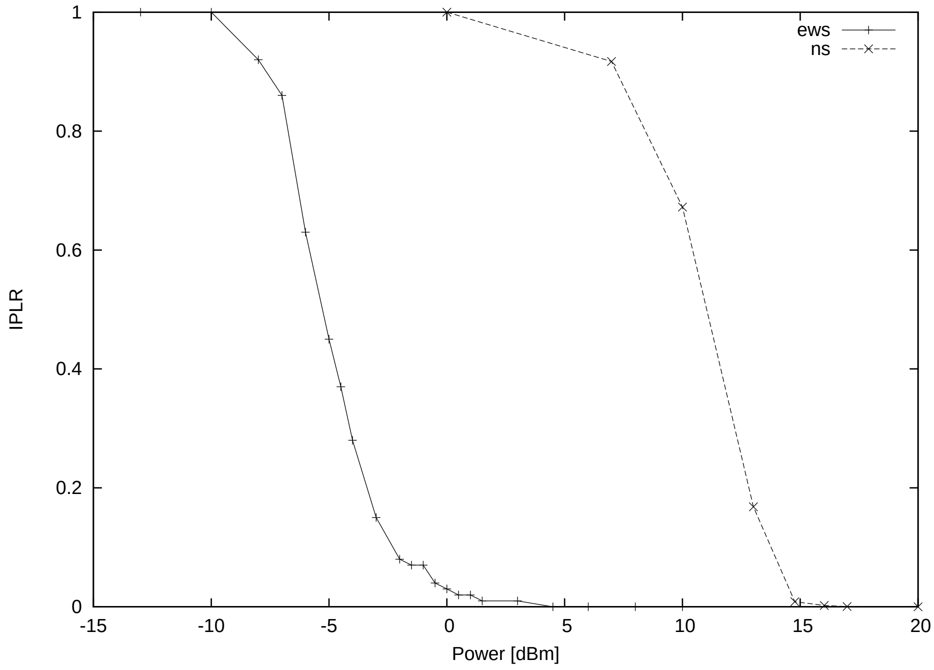
Testy pokazują, iż z punktu widzenia jakościowego symulatory *NS* i *EWS* są do siebie podobne - wyniki ilościowe jednak różnią się znacząco. Możliwa jest łatwa modyfikacja symulatora *EWS*, by jego wyniki zbliżyć do tych z *NSa*, jak na przykład zwiększyć czas transmisji ramki w scenariuszach 1. i 2., czy zmiana poziomu szumów tła, by wykresy ze scenariusza 3. dla modelu *TwoStateBerFunction* pokrywały się.



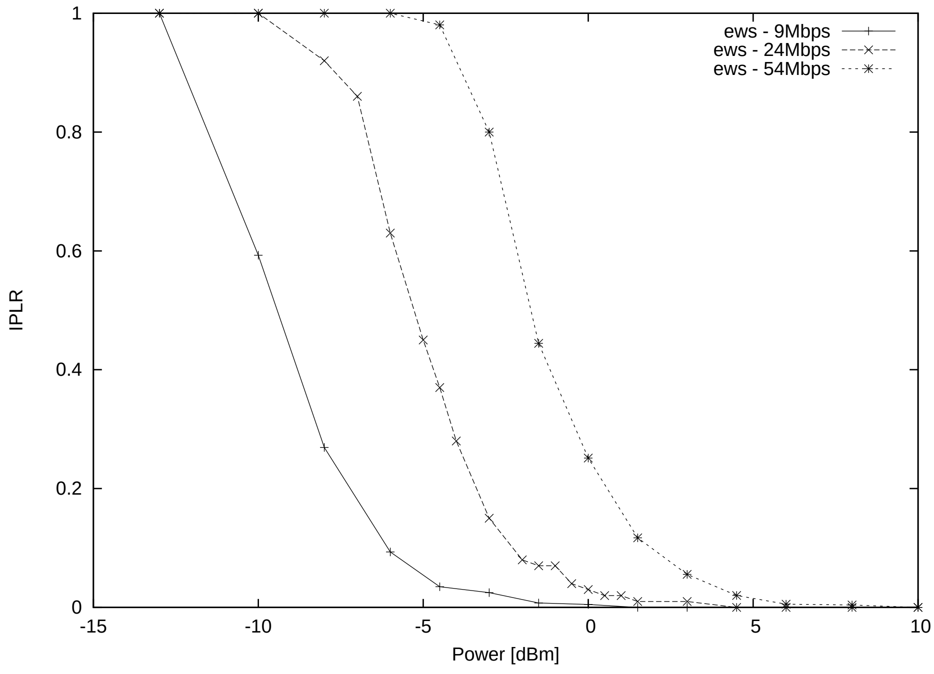
Rysunek 5.4: IPLR dla mocy nadawania 1dBm - model błędów *NS*



Rysunek 5.5: IPLR dla różnych mocy nadawania - model błędów *EWS*



Rysunek 5.6: IPLR dla prędkości transmisji 24Mbps



Rysunek 5.7: IPLR dla prędkości różnych prędkości transmisji w *EWS*

Rozdział 6

Podsumowanie

W pracy przedstawiony został proces projektowania oraz implementacji symulatora sieci bezprzewodowej *EWS*. Opisane zostały jego założenia, zarówno samego programu jak i mechanizmów odwzorowania prawdziwej sieci. Przedstawiona i omówiona została architektura symulatora. Zademonstrowany został także proces przygotowania prostej symulacji.

Przygotowany i przedstawiony został także zestaw testów, mający porównać działanie symulatora *EWS* z symulatorem *NS*. Testy te, w ujęciu jakościowym, przebiegły pomyślnie – zbadane zależności zgadzają się z naszymi przewidywaniami oraz z wynikami z symulatora *NS*. W ujęciu ilościowym widoczne są rozbieżności. Omówione zostały możliwe ich przyczyny.

6.1 Zalety i wady symulatora *EWS*

Udało nam się osiągnąć większość z zamierzonych celów. Wykorzystana architektura pozwala w jasny i przejrzysty sposób rozszerzać możliwości symulatora nie tylko o nowe protokoły, ale także modele fizyczne, modele błędów i modele propagacji.

Podczas symulacji przesyłane są rzeczywiste dane. Dzięki temu możliwe jest badanie procesów, które w typowym symulatorze zdarzeniowym są trudne lub niemożliwe do zaimplementowania.

Przygotowanie podstawowej symulacji jest procesem stosunkowo prostym, niewymagającym dobrej znajomości całego symulatora. Utworzony został także mechanizm ułatwiający przygotowanie skomplikowanych serii symulacji, włączając w to analizę danych wyjściowych oraz ich przedstawianie w rozsądny sposób.

EWS nie jest jednak pozbawiony wad. Pierwsza i być może najważniejsza z nich wynika z wykorzystanego języka: prędkość symulacji jest stosunkowo niska. W zależności od wykorzystanych protokołów, 1 sekunda symulowania odpowiada 1. milisekundzie czasu wewnętrznego symulacji. Oczywiście wraz z rozwojem sprzętu komputerowego czas ten będzie się zmniejszał.

Drugą wadą jest stosunkowo duże zużycie pamięci. Ze względu na model przesyłanych danych (opisany w punkcie 2.7) niezbędne jest tworzenie i przechowywanie dużej liczby obiektów BitBurst. Nie są one duże, gdyż zajmują ok. 50 bajtów + 1 bajt na każdy przesłany w danym cyklu symulacji bit. Jednak dla ramki o długości 5000 bitów, przy prędkości nadawania 56 Mbps i długości cyklu symulacji równemu czasowi przesłania 1 bitu niezbędne będzie utworzenie 5000 takich obiektów. Dla każdego odbierającego węzła tworzona jest kopia obiektu, więc przy jednym węźle nadającym i trzech odbierających liczba obiektów wzrasta do 20000. Odpowiada to prawie jednemu megabajtowi pamięci na każdą wysłaną ramkę. Oczywiście pamięć ta jest zwalniana, lecz jest to robione automatycznie i co jakiś czas, więc podczas symulacji zużycie pamięci sięga ponad 100 megabajtów.

Przyjęty model fizyczny jest dość ogólny i nie jest możliwe jego zastosowanie do przewidywania pokrycia rzeczywistych sieci, ze względu na brak uwzględnienia m.in. odbić czy jakichkolwiek interferencji. Z powodzeniem za to może być stosowany do badania wykorzystania sieci przy danym profilu ruchu, czy szacowania transferów osiąganych przez użytkowników. Może być także stosowany jako narzędzie do nauki mechanizmów funkcjonowania sieci.

6.2 Dalszy rozwój

Oprócz rozszerzania zbioru zaimplementowanych protokołów czy modeli błędów, przydatne może być wprowadzenie także kilku zmian w architekturze symulatora.

Pierwsza z nich to wprowadzenie mechanizmu pozwalającego pominąć te fragmenty symulacji, w których nie są wykonywane żadne działania, jak na przykład przeskakiwanie wszystkich czasów ciszy między nadawanymi ramkami (SIFS, DIFS, etc.), które mogą odpowiadać nawet kilkudziesięciu cyklom symulacji.

Drugie usprawnienie, to implementacja puli BitBurstów. Mogłaby ona zarówno zwiększyć prędkość symulacji, gdyż wykorzystywane byłyby wcześniej utworzone, nieużywane już obiekty i nie byłoby potrzeby tworzenia nowych, co jest czasochłonne. Ponadto, mogłaby zmniejszyć i ustabilizować ilość zużywanej pamięci.

Rozwijane mogą być także narzędzia związane z *EWS*. Przydatne byłoby dodanie do *ELVa* możliwości wizualizowania historii wysyłanych ramek/pakietów w postaci drabinki. *EWSStatistic* warto rozwinąć o bardziej zaawansowane metryki, by był jeszcze bardziej użyteczny.

Można także pójść w innym kierunku – zaimplementować model pozwalający na definiowanie i wykorzystanie łącz przewodowych. Umożliwiłoby to badanie efektywności działania access pointów, pozwalało np. zamodelować wpływ szerokości łącza internetowego na QOS użytkowników łączących się z siecią przez sieć bezprzewodową.

Warty rozważenia jest także graficzny interfejs umożliwiający przygotowanie symulacji bez potrzeby ingerencji w kod. Byłoby to jednak bardzo skomplikowane przedsięwzięcie, gdyż wymagałoby np. zdefiniowania formatu zapisu plików z symulacjami oraz ich obsługę.

Perspektywy rozwoju są szerokie. W zależności od potrzeb, symulator może wędrować w różnych kierunkach, które jednak, co ważne, nie są nawzajem sprzeczne.

Bibliografia

[1] Szczypiorski K. *Steganografia w bezprzewodowych sieciach lokalnych.* Politechnika Warszawska, 2006.

[2] The network simulator - ns-2. `http://isi.edu/nsnam/ns/`.

[3] Draft standard for floating-point arithmetic p754. [online], 2007. `http://www.validlab.com/754R/nonabelian.com/754/comments/Q754.129.pdf`.

[4] Sang Bae Kaixin Xu, Gerla M. How effective is the ieee 802.11 rts/cts handshake in ad hoc networks? [online], 2003. `http://www.cs.ucla.edu/NRL/wireless/uploads/kxu_globecom02.pdf`.

[5] Gnuplot. `http://www.gnuplot.info/`.

[6] Marc Greis. Tutorial for network simulator. `http://www.isi.edu/nsnam/ns/tutorial/index.html`.

[7] Opnet modeler wireless suite. `http://www.opnet.com/solutions/network_rd/modeler_wireless.html`.

[8] Global mobile information systems simulation library - glomosim. `http://pcl.cs.ucla.edu/projects/glomosim/`.

[9] Soltanian A. Golmie N., Van Dyck R. Interference of bluetooth and ieee 802.11: Simulation modeling and performance evaluation. [online], 2001. `http://www.antd.nist.gov/pubs/Golmiemswim01.pdf`.

[10] Gast M. *802.11 Wireless Networks: The Definitive Guide.* O'Reilly Media, 2002.

[11] Ieee std. 802.11, "wireless lan media access control (mac) and physical layer (phy) specifications. [online], 1999. `http://standards.ieee.org/getieee802/download/802.11b-1999.pdf`.

[12] Xiuchao W. Simulate 802.11b channel within ns2. [online], 2004. `http://www.comp.nus.edu.sg/~wuxiucha/research/reactive/publication/Simulate80211ChannelWithNS2.pdf`.

[13] Brenner B. *A Technical Tutorial on the IEEE 802.11 Standard.* Breezecom, 1997. `http://sss-mag.com/pdf/802_11tut.pdf`.

[14] Larry Leemis Steve Park. Discrete-event simulation: A first course. [online], 2000. `http://www.cs.wm.edu/~esmirni/Teaching/cs526/DESAFC-1.1.pdf`.

[15] W. David Kelton Averill M. Law. *Simulation Modeling and Analysis.* McGraw-Hill, 2000. Third Edition.

[16] George S. Fishman. *Discret-Event Simulation: Modeling, Programming, and Analysis.* Springer-Verlag, 2001. `http://books.google.com/books?id=PA3lj2KB3uEC`.

Data ostatniej aktualizacji adresów: 05.01.2010 r.

Wykaz skrótów

API	Application Programming Interface
BER	Bit Error Rate
CBR	Constant Bit Rate
CRC	Cyclic Redundancy Check
CVS	Concurrent Versions System
DCF	Distributed Coordination Function
DIFS	DCF Interframe Sequence
ELV	Ews Log Visualizer
EWS	Engineer's Wireless Simulator
FTP	File Transfer Protocol
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
HICCUPS	Hidden Communication System for Corrupted Networks
IDE	Integrated Development Environment
IPLR	IP Packet Loss Ratio
IPTD	IP Packet Transfer Delay
NS	Network Simulator
RTS/CTS	Request To Send / Clear To Send
SIFS	Short Interframe Sequence
SNR	Signal to Noise Ratio
TCL	Tool Command Language
TCP	Transfer Control Protocol
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System

Dodatek A

Zawartość płyty

Na płycie dołączonej do pracy znajdują się następujące elementy:

- katalog *engine2d* zawierający kod źródłowy projektu obsługującego grafikę 2d w programie *ELV*;
- katalog *EWS* zawierający kod źródłowy symulatora;
- katalog *EWSLogVisualizer* zawierający kod źródłowy projektu *EWS Log Visualizer*;
- katalog *EwsStatistic* zawierający kod źródłowy projektu *EWS Statistic*;
- katalog *NsCompareTests* zawierający kod źródłowy projektu z testami porównawczymi *EWS-NS*;
- katalog *NsScripts* zawierający skrypty tcl/perl/bash wykorzystywane do przeprowadzania symulacji porównawczych w symulatorze *NS*
- katalog *SampleLogs* zawierający kilka przykładowych plików wynikowych symulatora *EWS*;
- plik *ELV.jar* - wykonywalny plik Java Archive. Skompilowana i gotowa do uruchomienia wersja projektu *EWS Log Visualizer*;
- plik *EWSStatistic.jar* - wykonywalny plik Java Archive. Skompilowana i gotowa do uruchomienia wersja projektu *EWS Statistic*;

Katalogi *engine2d, EWS, EWSLogVisualizer, EwsStatistic, NsCompareTests* to elementy workspace z Eclipse. Możliwy jest ich import za pomocą Eclipse (file ▷ import ▷ select root directory ▷ wybór katalogu głównego płyty ▷ ok ▷ zaznaczenie projektów ▷ finish)