Technologie Sieciowe lista 3

Paweł Kubala

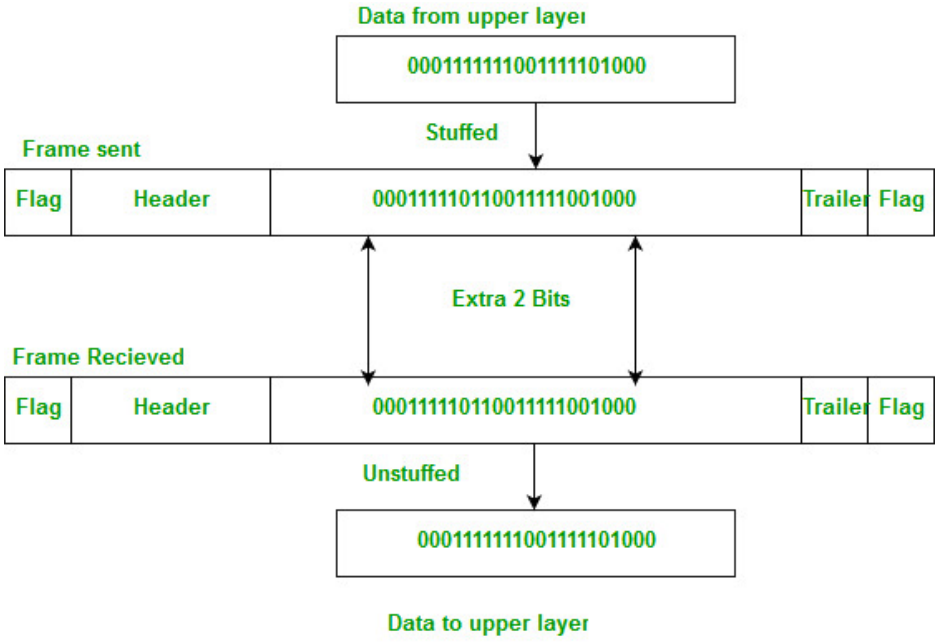
Zadanie 1

1.1 Problem

Napisz program ramkujący zgodnie z zasadą rozpychania bitów (podaną na wykładzie), oraz weryfikujacy poprawność ramki metodą CRC . Program ma odczytywać pewien źródłowy plik tekstowy 'Z' zawierający dowolny ciąg złożony ze znaków '0' i '1' (symulujacy strumień bitów) i zapisywać ramkami odpowiednio sformatowany ciąg do inngo pliku tekstowego 'W'. Program powinien obliczać i wstawiać do ramki pola kontrolne CRC – formatowane za pomocą ciągów złożonych ze znaków '0' i '1'. Napisz program, realizujacy procedure odwrotną, tzn. który odczytuje plik wynikowy 'W' i dla poprawnych danych CRC przepisuje jego zawartość tak, aby otrzymać kopię oryginalnego pliku źródłowego 'Z'.

1.2 Rozwiązanie

Zacznijmy od tego, jak powinno wyglądać ramkowanie:



Rysunek 1: Przykład ramkowania z rozpychaniem bitów

Na powyższym obrazku widzimy, że w celu stworzeniu ramki:

1. Rozpychamy bity naszej wiadomości,
2. Do tak sformatowanych danych dodajemy flagi – zazwyczaj jest to ciąg 01111110.

1.2.1 Rozpychanie bitów

Załóżmy, że chcemy wysłać wiadomość 01111110.
W celu wysłania danych potrzebujemy jakoś określić jej początek i koniec – uznajmy więc, że nasza `flaga` rozpoczynająca i kończąca wiadomość również jest ciągiem 01111110. Powstaje więc problem – spakowana ramka wygląda następująco: 011111100111111001111110 – odbiorca na podstawie pierwszego bajtu odczytuje nasze dane jako flagę kończącą wiadomość!
Aby zapobiec takim sytuacjom korzystamy właśnie z **bit stuffingu** – w naszym problemie przy napotkaniu 011111 wystarczy dodać dodatkowe 0 po ostatniej jedynce przed przepisywaniem dalszej części – w ten sposób rozwiązaliśmy problem przedwczesnego zakończenia wczytywania danych. Przykładowa implementacja tej metody wygląda następująco:

```
def bit_stuffing(s):
    result = '' # tutaj zapisujemy wynik
    ones_count = 0 # ile mamy jedynek pod rząd
    for c in s: # dla każdego bitu w wiadomości s
        result += c #dodajemy ten bit do wyniku
        if c == '1':
            ones_count += 1
            if ones_count == 5: #jeżeli trafiliśmy na 5 jedynkę pod rząd
                result += '0'# to dodajemy 0 do wyniku
                ones_count = 0
        else:
            ones_count = 0 # jak dostaliśmy 0, to zerujemy liczbę jedynek pod rząd
    return result
```

Oczywiście tak spakowane dane potrzebujemy też potrafić rozpakować – w tym przypadku postępujemy bardzo podobnie, jak przy rozpychaniu bitów, tylko zamiast dodać jedno zero po 5 piątkach – usuwamy je. Zaimplementujmy więc metodę odpakowywania:

```
def reverse_bit_stuffing(s):
    result = ''
    ones_count = 0
    for c in s:
        if c == '1':
            ones_count += 1
            result += c
        else:
            if ones_count < 5: #zero przepisujemy tylko wtedy, kiedy nie
                result += c    #jest ono poprzedzone przez 5 jedynek
            ones_count = 0
    return result
```

1.2.2 Cykliczna kontrola nadmiarowa

Mimo tego, że w teorii rozpychanie bitów powinno wystarczyć, to chcemy być pewni, że wiadomość zostanie poprawnie dostarczona do odbiorcy – w tym celu zastosujemy metodę CRC: do naszego bloku danych dodamy dodatkowo 32-bitową sekwencję kontrolną **FCS**. Zawiera ona nadmiarowe informacje, dzięki którym odbiorca będzie w stanie określić, czy dane zostały przesłane poprawnie.

W pythonie w celu skorzystania z `CRC` skorzystamy z gotowej funkcji biblioteki `zlib`. Implementacja funkcji zwracającej sekwencję FCS wygląda nastepująco:

```
def crc_check(s):
    crc = "{0:b}".format(zlib.crc32(s.encode()))
    return '0' *(32 - len(crc)) + crc
```

1.2.3 Tworzenie ramki

Posiadamy już wszystkie składniki potrzebne do zbudowania ramki – wystarczy połączyć to w całość:

1. ustawmy flagę 01111110 jako początek wiadomości
2. przygotujmy dane s:
   (a) FCS dla s dopisujemy na końcu danych
   (b) rozpychamy bity dla $s + FCS(s)$
3. tak przygotowane dane dodajemy do wiadomości
4. na zakończenie dodajemy flagę 01111110

Przykładowa implementacja w języku python:

```
def encode_frame(s):
    return '01111110' + bit_stuffing(s+crc_check(s)) + '01111110'
```

1.2.4 Wyciąganie danych z ramki

W celu uzyskania wiadomości z ramki postępujemy następująco:

1. wyciągnijmy 01111110 z początku i końca wiadomości.
2. Otrzymaną wiadomość odpakowywujemy metodą odwrotnego rozpychania bitów
3. wyciągamy ostatnie 32 bity z odpakowanej wiadomości – jest to sekwencja FCS
4. sprawdzamy, czy $FCS = CRC(s)$
   (a) jeżeli tak, to odczytujemy wiadomość
   (b) w przeciwnym wypadku dane przyszły uszkodzone

Przykładowa implementacja w języku python:

```
def decode_frame(s):
    decoded = reverse_bit_stuffing(s[8:(-8)])
    data = decoded[:len(decoded) - 32]
    crc = decoded[len(decoded) - 32:]
    if (crc_check(data) == crc):
        return data
    else:
        return ""
```

Zadanie 2

2.1 Problem

Napisz program (grupę programów) do symulowania ethernetowej metody dostepu do medium transmisyjnego (CSMA/CD). Wspólne łącze realizowane jest za pomocą tablicy: propagacja sygnału symulowana jest za pomoca propagacji wartości do sąsiednich komórek. Zrealizuj ćwiczenie tak, aby symulacje można było w łatwy sposób testować i aby otrzymane wyniki były łatwe w interpretacji.

2.2 Rozwiązanie

Zacznimy od tego, na czym polega protokół CSMA/CD – załóżmy, że mamy sieć składającą się z n komputerów. Każdy komputer może albo odbierać, albo wysyłać dane. W tym przypadku wszystkie urządzenia chcą wysłać swoje dane, co tworzy pewien problem - jeżeli dwa urządzenia wyślą w jednym momencie swoje dane, to między tymi danymi dojdzie do kolizji, na wskutek której zostaną one uszkodzone.
Rozwiązaniem tego problemu zajmuje się właśnie CSMA/CD, które działa następująco: dla każdego komputera:

1. sprawdź, czy jesteś gotowy do wysłania pakietów
2. jeżeli jesteś gotowy, to sprawdź, czy węzeł nie jest używany
   (a) jeżeli jest używany, to czekaj, aż będzie wolny
   (b) w przeciwnym wypadku zacznij nadawać swoje dane
3. sprawdź, czy nie nastąpiła kolizja
   (a) jeżeli wykryto kolizję, to poczekaj losową ilość czasu, a następnie znowu postaraj się wysłać wiadomość
4. przy braku kolizji zakończ sukcesem wysyłanie danych i zresetuj czas do ponownego wysłania pakietów.

2.2.1 Tworzenie klas obiektów

Wiemy już w jaki sposób działa nasz protokół. W celu zasymulowania go potrzebujemy 3 obiektów:

1. sieci,
2. komputerów, które będą odbierać i nadawać dane w sieci,
3. tablicy sygnałów odpowiadającej za rozprowadzanie danych po sieci.

2.2.1.1 Sygnały

Zacznijmy od stworzenia obiektu klasy pojedynczego sygnału. Składa się on z wysyłanych danych oraz kierunku, w którym te dane będą rozprowadzone:

```
class Sygnal():
    "klasa symulująca pojedynczy sygnał w sieci"
    #status 1 - wysyłaj w lewo
    #status 2 - wysyłaj w prawo
    #dane - dane w sygnale
    def __init__(self, status, dane):
        self.status = status
        self.dane = dane
```

2.2.1.2 Komputery

Mamy już sygnały, ale potrzebujemy też obiektów, które będą w stanie nadawać i odbierać te sygnały. Zdefiniujmy więc klasę `PC`, która będzie posiadać nastepujące elementy:

1. nazwę użytkownika,
2. położenie w sieci,
3. status wysyłania/nadawania,
4. dane do wysłania,
5. dane odebrane,
6. czas potrzebny do przgotowania wiadomości.

Implementacja tej klasy wygląda nastepująco:

```
class PC():
    def __init__(self, username, dane, interval, polozenie):
        #nazwa użytkownika
        self.username = username
        #status 0 - słucha
        #status 1 - wysyła
        self.status = 0
        #dane do wysłania
        self.dane = dane
        #położenie w sieci
        self.polozenie = polozenie
        #ile jeszcze będzie nasłuchiwał
        self.pauza = interval
        #lista danych, które odebraliśmy
        self.odebrane_dane = []
```

2.2.1.3 Sieć

Posiadamy już obiekty sygnałów i komputerów, więc potrzebujemy już tylko sieci. Składa się ona z nastepujących elementów:

1. długości przewodu,
2. listy poszczególnych jednostek przewodu, gdzie jedna jedostka składa się z listy sygnałów,
3. listy komputerów podłączonych do sieci,
4. czasu działania sieci.

Implementacja tej klasy wygląda następująco:

```
class Network():
    "Klasa symulująca naszą sieć"
    def __init__(self, dlugosc, komputery):
        #sygnały to tablica tablic zawierająca informacje o napięciach na naszej sieci
        self.sygnaly = [ [] for i in range(dlugosc) ]
```

```
        #komputery to nasi userzy, co nadają i odbierającsygnały
        self.komputery = komputery
        #długość sieci
        self.dlugosc = dlugosc
        #ile czasu minelo
        self.czas = 0
```

2.2.2 Wysyłanie sygnałów przez komputery

Zaimplementowaliśmy już wszystkie potrzebne klasy, więc czas zacząć pisać logikę. Głównym elementem programu jest wysyłanie sygnałów przez użytkowników, więc stwórzmy teraz funkcję, która dla każdego komputera w odpowiednim momencie wysyła do sieci dwa sygnały – jeden będzie rozprowadzał dane na lewo, a drugi na prawo od naszego PC.
Zaimplementujmy więc podstawową logikę z założeniem, że możemy nasłuchiwać jedynie części kabla, na której znajduje się nasz komputer:

```
    def wykonaj_akcje(self, sygnaly):
        "zwracanie: 0 - słuchaj, 1 - wysyłaj, 2 - wykryto kolizję"
        #zmniejszamy o 1 pozostały czas nasłuchiwania
        if self.pauza > 0 and self.status == 0:
            self.pauza -= 1

        #jeżeli nasłuchujemy i mamy pauzę, to nadal nasłuchujemy
        if self.pauza > 0 and self.status == 0:
            return 0

        #jeżeli nasłuchujemy i otrzymujemy jakiś sygnał, to nadal chcemy nasłuchiwać
        if self.status == 0 and len(sygnaly) > 0:
            return 0
        #w przeciwnym wypadku kończymy nasłuchiwać i zaczynamy wysyłać
        if self.pauza == 0 and self.status == 0:
            self.status = 1
            return 1
```

W powyższym kodzie brakuje jednak najważniejszego – detekcji kolizji. Pomyślmy więc jak rozwiązać ten problem. Zauważmy, że:

1. Podczas wysyłania wiadomości chcemy jakoś zablokować cały strumień – możemy to zrobić poprzez ciągłe wysyłania tych danych do momentu bycia pewnym, że kolizji nie było.

2. Jeżeli wyślemy do innego komputera sygnał i w trakcie wysyłania tego sygnału nie otrzymamy żadnego sygnału zewnętrznego to znaczy, że zdołaliśmy bez problemu nadać wiadomość.

3. Kolizję w najgorszym momencie wykrywamy najpóźniej w danej sytuacji:

   1) Załóżmy, że komputer A znajduje się na początku sieci, a komputer B na jej końcu. Sieć posiada n węzłów.

   2) A zaczyna wysyłać sygnał do B z prędkością węzeł/sekundę.

   3) Sekundę przed otrzymaniem sygnału – B wykrywa, że sieć jest czysta i wysyła swój sygnał.

   4) Dochodzi do kolizji.

5) Po kolejnych $n-1$ sekundach A otrzymuje informację, że nastąpiła kolizja i musi wysłać swoje dane po raz kolejny.

4. Czyli kolizja została wykryta po $2n$ sekundach. Czyli wystarczy, że nie wykryjemy kolzji przez $2x$(gdzie x to odległość komputera od dalszego krańca sieci) sekund aby być pewnym, że wiadomość przeszła bez przeszkód.

Na podstawie powyższych uwag możemy teraz dodać obsługę detekcji kolizji w naszym obiekcie `PC`. Dodatkowo teraz będzioemy potrzebować następujących elementów przy jego tworzeniu:

```
#odległość do krańca sieci
self.max_odleglosc = 0
#jak długo nadaje sygnał
self.aktualny_czas_nadawania = 0
#ile już było kolizji na tym PC
self.R = 0
```

Po dodaniu do konstruktora powyższych danych możemy uzupełnić naszą funkcję `wykonaj_akcje(self, sygnaly)` o następujący kod:

```
# jeżeli jesteśmy w trakcie wysyłania, to chcemy zobaczyć:
if self.status == 1 and len(sygnaly) == 0:
    self.aktualny_czas_nadawania += 1
    #1) Czy jesteśmy pewni, że nie było kolizji
    #jeżeli tak, to kończymy wysyłać nasze dane i wypisujemy sukces
    if self.aktualny_czas_nadawania == self.max_odleglosc * 2:
        print(self.username," rozesłał swoje dane!")
        self.status = 0
        self.aktualny_czas_nadawania = 0
        self.pauza = self.interval
        self.R = 0
        return 0
    else:
    #2) Nie jesteśmy pewni, że nie było kolizji - dalej wysyłamy te dane
        return 1
#jeżeli jesteśmy w trakcie wysyłania danych i odbierzemy
#jakieś inne dane, to znaczy, że mamy kolizję
if self.status == 1 and len(sygnaly) > 0 and sygnaly[0] != JAMMING_MESSAGE:
    self.status = 0
    self.aktualny_czas_nadawania = 0
    self.R += 1
    return 2
```

Teraz nasz komputer jest w stanie określić, kiedy ma wysłać sygnał i dodatkowo wykrywa

2.2.3 Wysyłanie sygnałów w sieci

`PC` jest już w stanie wysłać sygnał, więc dodajmy taką możliwość w sieci. W tym celu stworzymy metodę `wykonaj_akcje(self)` w klasie `Network`. Metoda ta będzie wykonywała 2 fukcje:

1. przesunięcie każdego sygnału w sieci,
2. wykonanie akcji dla kazdego obiektu `PC`, gdzie:

1) program kończymy jeżeli któryś z komputerów po raz 16 wykryje kolizję,
2) przy wykryciu kolizji użytkownik czeka pewną losową ilość czasu. Chcemy zmniejszyć prawdopodobieństwo kolejnej kolizji o połowę, dlatego za każdym razem losujemy z 2-krotnie większego przedziału, niż poprzednio.

Implementacja w pythonie wygląda nastepująco:

```
def wykonaj_akcje(self):
    self.czas += 1
    #przesuńmy wszystkie sygnały na tablicy
    sygnaly_temp = [ [] for i in range(self.dlugosc) ]
    for i in range(self.dlugosc):
        for sygnal in self.sygnaly[i]:
            if sygnal.status == 1 and i > 0:
                sygnaly_temp[i-1].append(sygnal)
            if sygnal.status == 2 and i+1 < self.dlugosc:
                sygnaly_temp[i+1].append(sygnal)

    self.sygnaly = sygnaly_temp
    # NARYSUJ AKTUALNY MODEL SIECI
    print(self.czas)
    for i in range(len(self.sygnaly)):
        print('[', end='')
        for s in self.sygnaly[i]:
            print(s.dane, end=' ')
        print(']', end='')
    print()
    for u in self.komputery:
        print(u.username," : ", u.odebrane_dane, " : ", u.ostatni_sygnal)
    print("_"*100)
    #wykonajmy akcję dla każdego użytkownika
    for user in self.komputery:
        akcja = user.wykonaj_akcje(self.sygnaly[user.polozenie])
        #akcja = 0 -> słucha
        #akcja = 1 -> wysyła
        #akcja = 2 -> kolizja
        if akcja == 1:
            self.sygnaly[user.polozenie].append(Sygnal(2, user.dane))
            self.sygnaly[user.polozenie].append(Sygnal(1, user.dane))
        if akcja == 2:
            # czekaj losową ilość czasu w danym zakresie
            user.pauza = random.randint(1, (2**(min(user.R, 10))-1)*self.dlugosc)
            if user.R >= 15:
                self.zakoncz = True
            self.sygnaly[user.polozenie].append(Sygnal(2, JAMMING_MESSAGE))
            self.sygnaly[user.polozenie].append(Sygnal(1, JAMMING_MESSAGE))
```

2.2.3 Odbieranie danych przez komputery

Potrafimy już wysyłać dane i stwierdzać, kiedy doszło do kolizji. Zostało nam określić, kiedy powinniśmy odbierać otrzymywane dane, a kiedy je kasować. Dodajmy sobie do naszej klasy `PC` zmienną `ostatni_sygnal`. Zmienna ta będzie przechowywać ostatnią wiadomość, która znajduje się na naszym strumieniu.
Dzięki dobrym zaimplementowaniu metody nadawania wiadomości jesteśmy w stanie szybko określić, kiedy powinniśmy otrzymaną wiadomość zapisywać, a kiedy kasować:

1. Jeżeli przez x czasu otrzymywaliśmy tę samą wiadomość i nagle przestaliśmy ją otrzymywać, a nie otrzymaliśmy też `JAMMING MESSAGE` to znaczy, że wiadomość została wysłana poprawnie i możemy ją zapisać.
2. Jeżeli otrzymaliśmy `JAMMING MESSAGE` to znaczy, że doszło do kolizji i wiadomość przechowywana w `ostatni_sygnal` jest nieważna.
3. Jeżeli na węźle mamy więcej, niż 1 sygnał, to doszło do kolizji - tutaj również wiadomość przechowywana w `ostatni_sygnal` jest nieważna.

Powyższe warunki dodajemy do metody wykonaj_akcje(self,sygnaly) w klasie PC:

```
        #jeżeli na naszym węźle jest tylko jeden sygnał, to chcemy go obsłużyć
        if len(sygnaly) == 1:
            #jeżeli ten sygnał to JAMMING_MESSAGE, to nasz ostatni sygnał
            #jest nieważny przez kolizję
            if sygnaly[0].dane == JAMMING_MESSAGE:
                 self.ostatni_sygnal = ''
            #jeżeli otrzymamy inny sygnał od JAMMING MESSAGE
            #i ostatni sygnał nie był pusty, to znaczy, że możemy odebrać ostatni sygnał
            elif sygnaly[0].dane != self.ostatni_sygnal and self.ostatni_sygnal != '':
                self.odebrane_dane.append(self.ostatni_sygnal)
                self.ostatni_sygnal = sygnaly[0].dane
            #jeżeli otrzymaliśmy nowe dane, a nasz ostatni sygnał był pusty, to
            #nasz nowy ostatni sygnał to otrzymane dane
            else:
                self.ostatni_sygnal = sygnaly[0].dane
        #jeżeli skończyliśmy otrzymywać sygnał, to jeżeli jakiś sygnał otrzymaliśmy, to
        #dodajemy go do listy
        elif len(sygnaly) == 0:
            if self.ostatni_sygnal != '':
                self.odebrane_dane.append(self.ostatni_sygnal)
                self.ostatni_sygnal = ''
        #jeżeli mamy kilka sygnałów naraz, to znaczy, że powstanie kolizja
        # więc nie bierzemy żadnego z nich
        else:
            self.ostatni_sygnal = ''
```

2.2.4 Odpalenie programu

Możemy teraz stworzyć przykładową sięc i zasymulować jej działanie:

```
def main():
    pc1 = PC("RYSIEK","01", 3, 0)
```

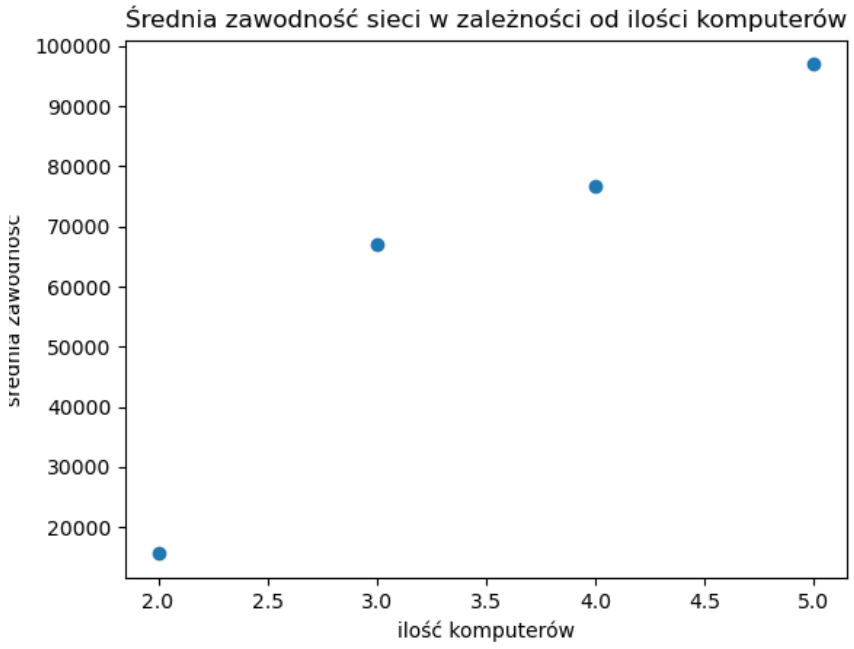
```
    pc2 = PC("BRAJAN", "10", 35, 10)
    pc3 = PC("MANIEK", "11", 23, 19)
    siec = Network(20,[pc1, pc2, pc3])
    while not siec.zakoncz:
        system("cls")
        siec.wykonaj_akcje()
        time.sleep(1)

if __name__ == "__main__":
    main()
```

2.2.5 Podsumowanie

Przeprowadziłem kilka symulacji dla różnych parametrów wejściowych. Brałem pod uwagę między innymi:

1. odległości od siebie poszczególnych komputerów,
2. ilość urządzeń w sieci,
3. pauzy między kolejnym wysyłaniem danych.

Dla porównania przy wywołaniu testów dla przypadku w 2.2.4 skuteczność(czyli ilość obrotów pętli przed osiągnięciem 16 kolizji przez jeden z komputerów) wynosi od 40000 do nawet 70000.
Po usunięciu *pc*3 wartości te zwiększyły zakres – od 55000 do 85000. Intuicyjnie możemy zobaczyć powód – im mamy więcej komputerów, tym większa szansa na to, że dwa z nich doprowadzą do kolizji. Przeprowadziłem więc 5 symulacji – każda posiada inną liczbę komputerów i każdy komputer wysyła sygnał co 5 sekund. Odległości między komputerami są równe i wynoszą 10, a każda symulacja została przeprowadzona 1000 razy.



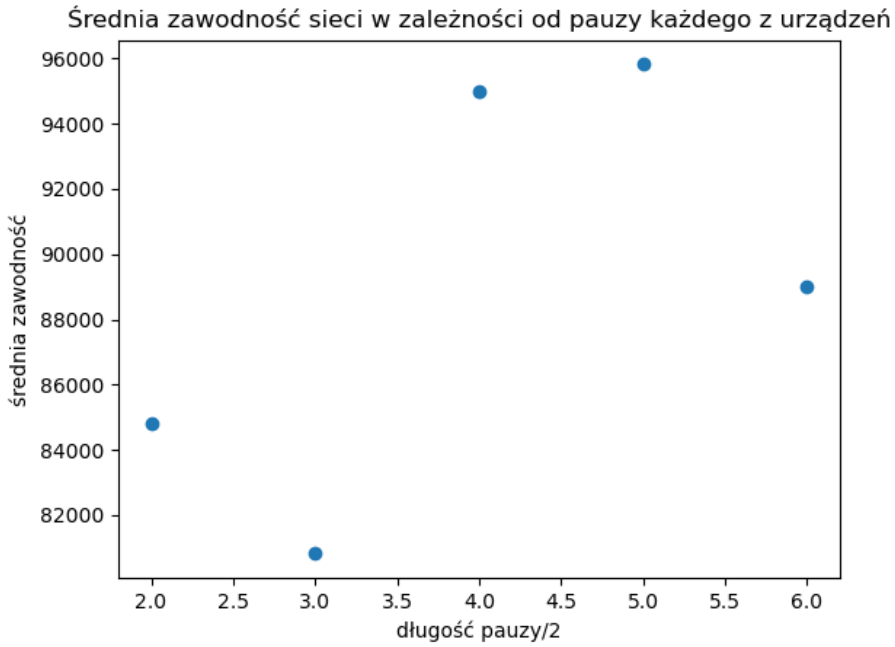
Rysunek 2: Wyniki symulacji w zależności od ilości komputerów

Jak widzimy - im więcej posiadamy komputerów w sieci, tym bardziej niezawodna jest sieć.

Dlaczego?

Ponieważ mimo tego, że ogólnie dochodzi do większej liczby kolizji, to są one rozproszone na coraz większą liczbę urządzeń oraz im więcej urządzeń, tym ciężej trafić na wolne łącze.

W przypadku zmiany czasów podczas symulacji średnie wyniki prezentują się następująco:

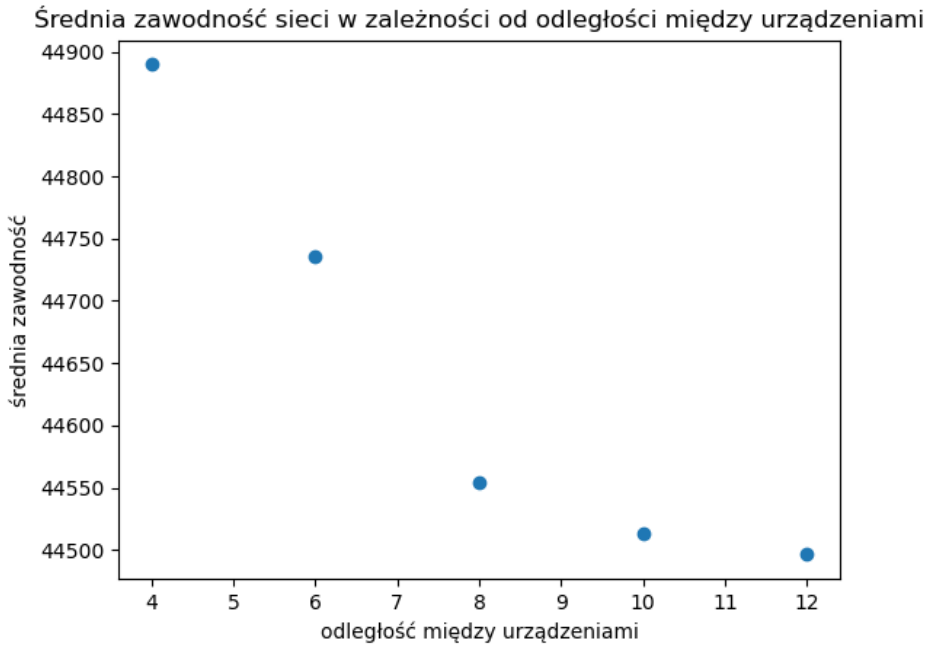


Rysunek 3: Wyniki symulacji w zależności od długości czekania komputerów

Jak widzimy – początkowy czas ma tak naprawdę małe znaczenie na niezawodność. Dlaczego? Ponieważ jak dojdzie do zderzenia w sieci, to oba urządzenia, między którymi doszło do kolizji czekają coraz więcej czasu, aż do 1024 obrotów pętli. Dodatkowo czas czekania jest również dobierany na podstawie długości sieci, dlatego dane mogą tak wyglądać.

Ostatnie co chcemy porównać, to odległości od siebie poszczególnych komputerów. Tutaj również bardzo łatwo wytłumaczyć, dlaczego im jest ona większa, tym gorsza jest niezawodność - im dalej dwa komputery znajdują się od siebie, tym więcej czasu potrzeba, aby jeden z nich zajął całą sieć. Ustawmy więc A i B. A jest na początku sieci, a B będziemy ustawiać coraz dalej od A.

Długość sieci to 15, a czas - 5.



Rysunek 4: Wyniki symulacji w zależności od odległości między komputerami