Lista 3

Technologie sieciowe

Patryk Majewski
250134

1 Wstęp

1.1 Opis zadania

W pierwszej części mamy za cel stworzenie programu umożliwiającego ramkowanie danych zgodnie z zasadą rozpychania bitów oraz odczyt danych z takich ramek.

Druga część zadania to zasymulowanie dostępu do medium transmisyjnego zgodnie z protokołem CSMA/CD.

1.2 Implementacja

Potrzebne programy zostały napisane w języku Python.

2 Ramkowanie

Ramki tworzone zgodnie z zasadą rozpychania bitów składają się z:

- bajtowych flag informujących o początku i końcu ramki
- nagłówka zawierającego min. informacje o adresacie i źródle
- faktycznych danych
- wartości kontrolnej CRC

Maksymalny rozmiar faktycznych danych w ramce jest zazwyczaj dosyć duży (jak na potrzeby zilustrowania mechanizmu rozpychania bitów), dlatego dla czytelności wyników ograniczymy tę liczbę do ∼32 bitów, tak żeby z niewielkiego pliku źródłowego uzyskać chociaż kilka ramek. Pominiemy szczegóły związane z nagłówkiem – dla uproszczenia założymy, że znajduje się on już w danych wejściowych.

Za flagę przyjmiemy 01111110, ponieważ jest ciągiem łatwym do wykrycia podczas kodowania danych.

2.1 CRC

Kod CRC wyznaczany jest w oparciu o dzielenie wielomianów nad ciałem $\mathbb{Z}_2$. Takie wielomiany możemy w łatwy sposób kodować za pomocą ciągów bitów, na przykład $x^3 + x\ + 1$ odpowiada 1011. Nadawca i adresat ustalają wspólnie $(r+1)$-bitowy generator G. Przyjmijmy też dodatkowe oznaczenia: D są naszymi danymi, a R to r-bitowy kod wyznaczony z pomocą generatora.

Podczas ramkowania wyznaczamy R korzystając ze wzoru

$$D \cdot 2^r = A\ \cdot G\ + R,$$

gdzie R jest resztą z dzielenia $\mathrm{D} \cdot 2^r$ przez G, pamiętając, że w $\mathbb{Z}_2$ odejmowanie równoznaczne jest dodawaniu i właściwie oznacza operację xor.

Pseudokody:

```
def division_remainder(x: str):
    next = len(G)
    R = x[:next]
    while next ⩽ len(x):
        if R[0] = '1':
            R = R xor G
        R = R[1:]
        if next = len(x):
            break
        R += x[next]
        next += 1
    return R
```

```
def CRC(x: str):
    x += '0' * (len(G) - 1)
    return division_remainder(x)
```

Na przykład, dla G = 1001 i D = 101110 procedura CRC ma następujący przebieg:

$$
\begin{array}{rcccccccccl}
 & & & & 1 & 0 & 1 & 0 & 1 & 1 & \leftarrow A \\
D \rightarrow & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & \div\, 1001 \leftarrow G \\
 & 1 & 0 & 0 & 1 & & & & & & \\
\cline{4-10}
 & & & 1 & 0 & 1 & 0 & & & & \\
 & & & 1 & 0 & 0 & 1 & & & & \\
\cline{5-10}
 & & & & & 1 & 1 & 0 & 0 & & \\
 & & & & & 1 & 0 & 0 & 1 & & \\
\cline{6-10}
 & & & & & & 1 & 0 & 1 & 0 & \\
 & & & & & & 1 & 0 & 0 & 1 & \\
\cline{7-10}
 & & & & & & 0 & 0 & 1 & 1 & \leftarrow R
\end{array}
$$

W ramce umieszczamy $D \cdot 2^r + R$, zatem w naszym przykładzie byłoby to 101110 011.

Aby sprawdzić poprawność danych, odbiorca musi tylko podzielić (w $\mathbb{Z}_2$) sekwencję między flagami przez G. Jeśli nie otrzyma reszty, dotarła poprawna wiadomość. W przeciwnym wypadku ramka zostaje odrzucona.

W naszym modelu wybierzemy G jak w przykładzie, 1001.

2.2 Kodowanie

Dane do przesłania zamieniamy na ramki w następujący sposób:

1. Czytamy 32 kolejne bity danych wejściowych (jeśli to możliwe – gdy jest ich mniej, czytamy wszystkie).
2. Obliczamy kod CRC i dostawiamy go na koniec naszego ciągu.
3. Przeglądamy nasz ciąg i po każdej sekwencji pięciu jedynek dostawiamy zero, żeby pozbyć się z niego fałszywych flag.
4. Dodajemy flagi na początku i końcu.
5. Zapisujemy utworzoną ramkę, a następnie – jeśli nie skończyły się dane wejściowe – tworzymy kolejną, wracając do kroku pierwszego.

2.3 Dekodowanie

Odpakowywanie danych z ramki wykonywane jest następująco:

1. Odczytujemy fragment między dwiema flagami.
2. W otrzymanym ciągu usuwamy zera występujące po sekwencji pięciu jedynek.
3. Dzielimy uzyskany ciąg przez G. Jeśli otrzymamy resztę, ramka uległa uszkodzeniu i jest odrzucana.
4. Jeśli ramka jest poprawna, usuwamy z końca kod CRC (przy naszym G są to trzy bity).
5. Zapisujemy rezultat i, jeśli mamy więcej fragmentów, wracamy do punktu pierwszego.

2.4 Testy

2.4.1 Podstawowa funkcjonalność

Zacznijmy od przypadku z przykładu:

```
>>> x = encode('101110')

>>> print(x)
    0111111010111001101111110

>>> decode(x)
    '101110'
```

Przykład, gdzie musimy dopisać zera:

```
>>> x = encode('01111110')

>>> print(x)
    0111111001111101000001111110

>>> decode(x)
    '01111110'
```

2.4.2 Test z plikiem

Z.txt

```
10011111 10010010 10001011 11001110 00110001 00011000 01111111 10110100 01010010 01110001
10010100 10111001 11010001 11111100 10011011
01101001 00101100 11100010 10111111 11111001 00000101 10000000
```

W.txt

```
0111111010011111010010010100010111100111011001111110 0111111000110001000110000111110111011010010101111110
0111111001010010011100010100111110
0111111010010100101110011101000111110110010101111110 0111111010011011010011111110
0111111001101001001011001110001010111110101101111110 0111111011111000100000101100000000111011111110
```

Z1.txt

```
10011111 10010010 10001011 11001110 00110001 00011000 01111111 10110100 01010010 01110001
10010100 10111001 11010001 11111100 10011011
01101001 00101100 11100010 10111111 11111001 00000101 10000000
```

```
def test_decoding():
    create_frames("Z.txt", "W.txt")
    decode_frames("W.txt", "Z1.txt")
    with open("Z.txt") as f1, open("Z1.txt") as f2:
        for l1, l2 in zip(f1, f2):
            if l1 != l2:
                break
        else:
            print("Success!")

>>> from zad1 import test_decoding
>>> test_decoding()
    Success!
```

2.4.3 Błąd wewnątrz danych

Spróbujemy teraz zmienić któryś z bitów w pliku W.txt przed jego odkodowaniem.

```
Z.txt
10011111 10010010 10001011 11001110 00110001 00011000 01111111 10110100 01010010 01110001
10010100 10111001 11010001 11111100 10011011
01101001 00101100 11100010 10111111 11111001 00000101 10000000

popsuty W.txt
011111101000011101001001010001011110011101100111111 0111111000110001000110000111110111011010010101111110
0111111001010010011100010100111110
...

Z1.txt
00110001 00011000 01111111 10110100 01010010 01110001
10010100 10111001 11010001 11111100 10011011
01101001 00101100 11100010 10111111 11111001 00000101 10000000


>>> from zad1 import create_frames, decode_frames
>>> create_frames("Z.txt", "W.txt")

*psujemy plik W.txt*

>>> decode_frames("W.txt", "Z1.txt")
    Warning: Frame containing 1000011101001001010001011110011101100111101 omitted.

>>> with open("Z.txt") as f1, open("Z1.txt") as f2:
        for l1, l2 in zip(f1, f2):
            if l1 != l2:
                print("Failure.")
                break
        else:
            print("Success!")

    Failure.
```

Ramka zawierająca błąd została całkowicie odrzucona. Cała reszta wyników jest poprawna.

2.4.4 Błąd we fladze

W ostatnim teście w pierwszej linijce pliku wynikowego zepsujemy jedną z flag:

```
Z.txt
10011111 10010010 10001011 11001110 00110001 00011000 01111111 10110100 01010010 01110001
...

popsuty W.txt
011111101001111101001001010001011110011101101111010 0111111000110001000110000111110111011010010101111110
0111111001010010011100010100111110
...

Z1.txt
00110001 00011000 01111111 10110100 01010010 01110001
...
```

Kod jest identyczny jak w poprzednim teście. Również w tym przypadku cała uszkodzona ramka została odrzucona, natomiast reszta danych odczytana jest poprawnie.

3 Protokół CSMA/CD

3.1 Wstępne ustalenia

W protokole CSMA/CD węzeł sieci rozpoczyna transmisję, gdy wykryje, że łącze jest wolne. Może się okazać, że dwa węzły rozpoczną transmisję w podobnym momencie, co spowoduje kolizję. Dlatego też transmitujący węzeł na bieżąco monitoruje medium. Jeśli odkryje, że sygnał na łączu różni się od tego, który nadaje, rozpoczyna procedurę rozwiązywania konfliktu:

1. Przestaje transmitować właściwe dane. Zamiast tego wysyła jam signal, który poinformuje inne węzły o kolizji i pozwoli im odrzucić nadawaną ramkę przez wywołanie błędu CRC.
2. Wybiera z rozkładem jednostajnym czas oczekiwania ze zbioru $\{0, \ ..., \ 2^n - 1\}$, gdzie $n = min(c, 10)$, a c jest liczbą kolizji które wystąpiły podczas prób transmisji obecnej ramki.
3. Po odczekaniu ustalonego wcześniej czasu podejmowana jest ponowna próba transmisji.

Nadawca uznaje ramkę za poprawnie wysłaną w momencie, gdy kończy transmisję bez wykrycia kolizji. Ważne jest zatem ustalenie minimalnego rozmiaru ramki, tak żeby w przypadku kolizji każda transmitująca stacja miała szansę się o niej dowiedzieć.

3.2 Założenia

Symulacja ma turowy przebieg, zatem jedną iterację (wywołanie metody `step`) uznajemy za podstawową jednostkę czasu. Rozmiar danych wyrażać będziemy we fragmentach. W jednej turze węzeł jest w stanie wyemitować maksymalnie jeden taki fragment.

Przy inicjalizacji modelu podawany jest rozmiar przewodu R (tj. liczba jego segmentów). Dane poruszają się po przewodzie z prędkością 1 segment na iterację.

Do każdego z segmentów możemy podłączyć maksymalnie jedno urządzenie. Monitorowanie medium przez węzeł w celu wykrycia kolizji odbywa się poprzez obserwację segmentu przewodu, do którego jest podłączony.

Wspomniany wcześniej minimalny rozmiar ramki danych wybierzemy po rozpatrzeniu przypadku skrajnego:

1. Węzeł A na jedym końcu przewodu wysyła sygnał.
2. Jedną iterację przed tym, jak sygnał dotarłby do węzła B na drugim końcu przewodu (czyli po upływie kolejnych R-2 iteracji), również zaczyna on transmisję.

B zaczyna emitować jam signal w następnej turze. Aby A miał szansę na odkrycie kolizji, potrzebuje emitować sygnał jeszcze przez R-2 iteracji. Sygnał musi być więc nadawany łącznie przez przynajmniej 2R-2 iteracji, czyli mieć rozmiar przynajmniej 2R-2 fragmentów.

Dodatkowe szczegóły dotyczące implementacji zostały umieszczone w dokumentacji w pliku źródłowym.

3.3 Użycie

Model konstruujemy poprzez stworzenie instancji klasy `Simulation` (za argument podając R), a następnie podpięcie urządzeń z użyciem metody `add_node`. Metoda przyjmuje za argumenty kolejno:

- nazwę urządzenia (w wyświetlanych wynikach aliasowaną do pierwszej litery, dlatego na przykład niewskazane są urządzenia o nazwach K1 i K2)
- segment przewodu, do którego chcemy się podłączyć (od 0 do R-1)
- numer iteracji, w której urządzenie rozpocznie aktywność
- liczbę ramek, które urządzenie ma przesłać w czasie trwania symulacji

Metoda uruchamiająca symulację to `run(output_all: bool, display_time: float)`. Dostępne tryby:

- bez argumentów – po każdym kroku wyświetlany jest obecny stan, program kontynuuje działanie po wciśnięciu klawisza Enter

- `output_all=True` – drukowany jest od razu cały ślad stanów symulacji
- podany `display_time` – podobnie jak tryb domyślny, ale zamiast oczekiwać na Enter, program kontynuuje po podanym czasie

Przykładowy setup:

```
>>> from zad2 import Simulation
>>> sim = Simulation(5)
>>> sim.add_node('A', 0, 0, 1)
>>> sim.add_node('B', 2, 0, 0)
>>> sim.add_node('C', 4, 3, 1)
>>> sim.run()
```

Wybrana klatka przykładu:

```
+------+----------+----------+----------+----------+----------+
| Name |    A     |          |    B     |          |    C     |
| Wait |    4     |          |    0     |          |    7     |
| Flag |  Active  |          |   Idle   |          | Collided |
+======+==========+==========+==========+==========+==========+
| t=4  |    A     |    A     |    A     |   A, C   |  A, C*   |
+------+----------+----------+----------+----------+----------+
A continues to broadcast.
C has detected a collision. It started to broadcast its jam signal.
C continues to broadcast.
Press Enter to continue...
```

Górny wiersz pokazuje stan urządzeń, dolny reprezentuje połączenie. Pierwsza komórka dolnego wiersza informuje o numerze iteracji. Sygnał wysłany przez dany węzeł reprezentowany jest pierwszym znakiem jego nazwy. Dodatkowa gwiazdka symbolizuje jam signal.

Symulacja kończy się w momencie, gdy urządzenia nie planują już niczego wysyłać, a przez przewód nie przechodzą żadne sygnały. Wyświetlany jest wówczas komunikat informujący o podstawowych statystykach:

```
108 iterations total. Node statistics:

A
collisions: 5
total waiting time: 7

B
collisions: 0
total waiting time: 0

C
collisions: 5
total waiting time: 36
```