Technologie Sieciowe – LISTA 3

Lena Jasińska 261740

Maj 2022

# Wstęp

Zadanie pierwsze polega na napisaniu programu kodującego i dekodującego dane przez ramkowanie, zgodnie z zasadą rozpychania bitów.

Zadanie drugie polega na stworzeniu symulacji dostępu do medium transmisji (zgodnie z protokołem CSMA/CD).

# Zadanie pierwsze – ramkowanie

Budowa ramki:

kod CRC (tu: 32 bity)

dane (zmodyfikowane rozpychaniem bitów)

flaga początku

flaga końca

* Za flagi początki i końca przyjmujemy 01111110.
* Przesyłane dane są zmodyfikowane w taki sposób, żeby zawsze po sekwencji 5 jedynek następne było 0. W ten sposób nie będzie w nich fałszywych flag.
* Kod CRC pozwala nam na weryfikację poprawności odczytanych danych. Korzystamy z CRC32, czyli ma on 32 bity.

## Kodowanie:

1. Czytamy z wejścia *frameSize* liczbę bitów (w większości testów *frameSize* = 50) – lub mniej, jeśli zostało mniej bitów.
2. Liczymy kod CRC i dostawiamy go na końcu ciągu.
3. Zamieniamy wszystkie „11111” na „111110”.
4. Na początek i koniec dodajemy flagi.
5. Powtarzamy, dopóki są bity na wejściu.

## Dekodowanie:

1. Czytamy fragment danych między dwiema flagami.
2. Zamieniamy wszystkie „111110” na „11111”.
3. Liczymy kod CRC z otrzymanych danych (bez 32 ostatnich bitów) i porównujemy z kodem CRC zapisanym na ostatnich 32 bitach.
4. Jeśli są równe, ramka jest poprawna – usuwamy z końca kod crc i dopisujemy do wyniku.
5. Jeśli są różne, ramka jest uszkodzona – odrzucamy ją.
6. Powtarzamy, dopóki są ramki na wejściu.

## Testy:

* Przykładowy wynik działania programu:

Input size: 98, input: 01110010100111111111111101010010101010110100101010101010101000110101010101010101010101010101010101

KODOWANIE:

1) data: 01110010100111111111111101010010101010110100101010

CRC: 11010110111001100011111011000101

Frame: 01111110011100101001111101111101110101001010101011010010101011010110111001100011111001100010101111110

2) data: 101010101000110101010101010101010101010101010101

CRC: 10101000110100000100010111110011

Frame: 0111111010101010100011010101010101010101010101010101010110101000110100000100010111110001101111110

Coded: 011111100111001010011111011111011101010010101010110100101010110101101110011000111110011000101011111100111

111010101010100011010101010101010101010101010101010110101000110100000100010111110001101111110

DEKODOWANIE:

Ramka nr 1

crc read: 11010110111001100011111011000101, crc calculated: 11010110111001100011111011000101

Decoding - success: 01110010100111111111111101010010101010110100101010

Ramka nr 2

crc read: 10101000110100000100010111110011, crc calculated: 10101000110100000100010111110011

Decoding - success: 101010101000110101010101010101010101010101010101

Decoded: 01110010100111111111111101010010101010110100101010101010101000110101010101010101010101010101010101

DECODED = ORIGINAL

input-decoded.txt:

01110010 10011111 11111111 01010010 10101011 01001010 10101010 10100011 01010101 01010101 01010101 01010101 01

01110010 10011111 11111111 01010010 10101011 01001010 10101010 10100011 01010101 01010101 01010101 01010101 01

input.txt:

01111110 01110010 10011111 01111101 11010100 10101010 11010010 10101101 01101110 01100011 11100110 00101011 11110011 11110101 01010100 01101010 10101010 10101010 10101010 10101101 01000110 10000010 00101111 10001101 111110

input-coded.txt:

* Spróbujemy teraz przeczytać uszkodzoną ramkę. W tym celu edytujemy plik input-coded.txt:

input-coded.txt:

01111110 01111010 10011111 01111101 11010100 10101010 11010010 10101101 01101110 01100011 11100110 00101011 11110011 11110101 01010100 01101010 10101010 10101010 10101010 10101101 01000110 10000010 00101111 10001101 111110

* Program wykryje błąd danych, odrzucając uszkodzoną ramkę. Pozostałe ramki zostaną przeczytane poprawnie.

DEKODOWANIE:

Ramka nr 1

crc read: 11010110111001100011111011000101, crc calculated: 10101001001011111010010101010110

Decoding - FAIL.

Ramka nr 2

crc read: 10101000110100000100010111110011, crc calculated: 10101000110100000100010111110011

Decoding - success: 101010101000110101010101010101010101010101010101

Decoded: 101010101000110101010101010101010101010101010101

DECODED != ORIGINAL

* Podobnie będzie w przypadku, gdy błąd wystąpi w pierwszej/ostatniej fladze lub w CRC.

01111010 01110010 10011111 01111101 11010100 10101010 11010010 10101101 01101110 01100011 11100110 00101011 11110011 11110101 01010100 01101010 10101010 10101010 10101010 10101101 01000110 10000010 00101111 10001101 111110

input-coded.txt:

DEKODOWANIE:

Ramka nr 1

crc read: 11010110111001100011111011000101, crc calculated: 11001110001100011011110101111111

Decoding - FAIL.

Ramka nr 2

crc read: 10101000110100000100010111110011, crc calculated: 10101000110100000100010111110011

Decoding - success: 101010101000110101010101010101010101010101010101

Decoded: 101010101000110101010101010101010101010101010101

DECODED != ORIGINAL

01111110 01110010 10011111 01111101 11010100 10101010 11010010 10101101 01101110 01100011 11100110 10101011 11110011 11110101 01010100 01101010 10101010 10101010 10101010 10101101 01000110 10000010 00101111 10001101 111110

DEKODOWANIE:

Ramka nr 1

crc read: 11010110111001100011111011010101, crc calculated: 11010110111001100011111011000101

Decoding - FAIL.

Ramka nr 2

crc read: 10101000110100000100010111110011, crc calculated: 10101000110100000100010111110011

Decoding - success: 101010101000110101010101010101010101010101010101

Decoded: 101010101000110101010101010101010101010101010101

DECODED != ORIGINAL

input-coded.txt:

* Jeśli błąd wystąpi w fladze początku/końca pomiędzy dwoma ramkami, obie będą błędne, ale pozostałe ramki zostaną odczytane poprawnie (tu mniejszy *frameSize* = 30)

Były 4 ramki. Ramki 1 i 2 zostały potraktowane jako jedna błędna ramka i odrzucone. Kolejne ramki zdekodowane prawidłowo. (to zależy od implementacji)

DEKODOWANIE:

Ramka nr 1

crc read: 10010010010010011011001110001000, crc calculated: 01101011101001010010110110100011

Decoding - FAIL.

Ramka nr 2

crc read: 10110110111100000011111011001010, crc calculated: 10110110111100000011111011001010

Decoding - success: 001101010101010101010101010101

Ramka nr 3

crc read: 00101101110110101111010110100110, crc calculated: 00101101110110101111010110100110

Decoding - success: 01010101

Decoded: 00110101010101010101010101010101010101

DECODED != ORIGINAL

input-coded.txt:

01111110 01110010 10011111 01111101 11010100 11110001 00100111 11000111 00001001 00101111 00111111 01010101 01101001 01010101 01010101 00100100 10010011 01100111 00010000 11111100 11111100 01101010 10101010 10101010 10101101 10110111 10000001 11110011 00101001 11111001 11111001 01010100 10110111 01101011 11010110 10011001 111110

# Zadanie drugie – protokół CSMA/CD

## Protokół:

Protokół CSMA/CD (*Carrier Sense Multiple Access with Collision Detection*) służy do zarządzania problemem kolizji w przypadku korzystania przez wiele węzłów ze wspólnego medium transmisyjnego. Węzły monitorują medium – gdy wykryją, że jest wolne, mogą zacząć nadawać. Jeśli jednak inny węzeł w bliskim czasie także zacznie nadawać (odstęp czasowy zbyt mały, by mogły się wzajemnie wykryć), dojdzie do kolizji.

Logika wykrywania kolizji przez węzeł:

1. Jeśli nadaję sygnał, a sygnał w medium różni się od tego, co nadaję, nastąpiła kolizja.
2. Zaczynam wysyłać jam signal w celu poinformowania innych węzłów o kolizji.
3. Muszę nadać to, co zacząłem – losuję czas oczekiwania między 0 a 2n – 1, gdzie n to liczba kolizji przy wysyłaniu obecnej ramki, max 10. Mnożę razy czas propagacji.
4. Po zakończeniu nadawania jam signal, odczekuję zadany czas i ponawiam próbę transmisji.

## Długość ramki:

Do wykrycia kolizji przez nadający węzeł niezbędna jest odpowiednia długość ramki. Musi być ona na tyle duża, by czas jej wysłania był dwukrotnością czasu propagacji w medium (minus 2).

Przykład:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | a | a | a | a | a | a | a | a | b |

A B

W powyższym przypadku A nadaje sygnał od n-1 = 9 kroków. B zaczyna nadawać sygnał, bo jeszcze nie widzi, że medium jest zajęte (sygnał jeszcze do niego nie doszedł). W następnym kroku sygnały na 9 i 10 polu zostaną zepsute i B dowie się o kolizji.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | a | a | a | a | a | a | a | ab | ba |

A B

Żeby A dowiedział się o kolizji, zepsuty sygnał musi przejść jeszcze n-2 = 8 kroków, a A musi wciąż nadawać. To oznacza, że A musi nadawać minimum przez (n – 1) + 1 + (n – 2) = 2n – 2 kroków.

## Symulacja:

Uruchomienie: *java Zad2 [size] [time] [probability] [positionA] [positionB] [positionC] ... itd., max A-Z, position od 1*

Medium transmisyjne jest reprezentowane w symulacji przez tablicę sygnałów o podanej wielkości *size*. Wartość *time* określa liczbę iteracji, przez które wykonywać się będzie symulacja. Wartość *probability* określa prawdopodobieństwo, z jakim nieaktywny węzeł będzie chciał zacząć nadawać w danej iteracji. Dalej podajemy pozycje (od 1), do których są podpięte poszczególne węzły.

Na symulację składają się 3 klasy: Network, Node i Signal.

W klasie Network znajduje się główna pętla symulacji, simulate(). Każda iteracja imituje jedną jednostkę czasu, w jakiej sygnał przesuwa się o jeden segment w tablicy (emisja także następuje z prędkością 1 sygnał / iterację).

## Wizualizacja:

Wyświetlane informacje pozwalają nam bez trudu śledzić przepływające w medium informacje. Widzimy zarówno numer iteracji, jak i stan medium, oraz akcje podejmowane aktualnie przez węzły. Możemy śledzić przepływ danych, kolizje oraz reakcje węzłów na kolizje. Przy opisie akcji węzła pierwsza liczba w [] oznacza liczbę kolizji, które wystąpiły jak dotąd przy nadawaniu obecnej ramki. Druga z liczb (w przypadku akcji „jamming”) oznacza wylosowany czas (w iteracjach), jaki węzeł zamierza odczekać przed ponowną próbą wysłania ramki, której nie udało mu się wysłać.

Po zakończeniu symulacji jest wyświetlane także krótkie podsumowanie dla każdego węzła – liczba wysłanych z sukcesem wiadomości oraz liczba wszystkich wykrytych kolizji.

## Fragmenty przykładowych uruchomień:

Przykład 1:

time 57: [\_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_]

time 58: [\_\_\_\_\_, <A> , \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_] |A: talking [0]

time 59: [ <A , <A> , A> , \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_] |A: talking [0]

time 60: [ <A , <A> , A> , A> , \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_] |A: talking [0]

time 61: [ <A , <A> , A> , A> , A> , \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_] |A: talking [0]

time 62: [ <A , <A> , A> , A> , A> , A> , \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_] |A: talking [0]

time 63: [ <A , <A> , A> , A> , A> , A> , A> , \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_] |A: talking [0]

time 64: [ <A , <A> , A> , A> , A> , A> , A> , A> , \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_] |A: talking [0]

time 65: [ <A , <A> , A> , A> , A> , A> , A> , A> , A> , \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, <C> ] |A: talking [0] |C: talking [0]

time 66: [ <A , <A> , A> , A> , A> , A> , A> , A> , A> , A> , \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, <C , <C> ] |A: talking [0] |C: talking [0]

time 67: [ <A , <A> , A> , A> , A> , A> , A> , A> , A> , A> , A> , \_\_\_\_\_, <C , <C , <C> ] |A: talking [0] |C: talking [0]

time 68: [ <A , <A> , A> , A> , A> , A> , A> , A> , A> , A> , A> , <AC>, <C , <C , <C> ] |A: talking [0] |C: talking [0]

time 69: [ <A , <A> , A> , A> , A> , A> , A> , A> , A> , A> , <AC>, <AC>, <AC>, <C , <C> ] |A: talking [0] |C: talking [0]

time 70: [ <A , <A> , A> , A> , A> , A> , A> , A> , A> , <AC>, <AC>, <AC>, <AC>, <AC>, <C> ] |A: talking [0] |C: talking [0]

time 71: [ <A , <A> , A> , A> , A> , A> , A> , A> , <AC>, <AC>, <AC>, <AC>, <AC>, <AC>, <X> ] |A: talking [0] |C: jamming [1, 0]

time 72: [ <A , <A> , A> , A> , A> , A> , A> , <AC>, <AC>, <AC>, <AC>, <AC>, <AC>, <X> , <X> ] |A: talking [0] |C: jamming [1, 0]

time 73: [ <A , <A> , A> , A> , A> , A> , <AC>, <AC>, <AC>, <AC>, <AC>, <AC>, <X> , <X> , <X> ] |A: talking [0] |C: jamming [1, 0]

time 74: [ <A , <A> , A> , A> , A> , <AC>, <AC>, <AC>, <AC>, <AC>, <AC>, <X> , <X> , <X> , <X> ] |A: talking [0] |C: jamming [1, 0]

time 75: [ <A , <A> , A> , A> , <AC>, <AC>, <AC>, <AC>, <AC>, <AC>, <X> , <X> , <X> , <X> , <X> ] |A: talking [0] |C: jamming [1, 0]

time 76: [ <A , <A> , A> , <AC>, <AC>, <AC>, <AC>, <AC>, <AC>, <X> , <X> , <X> , <X> , <X> , <X> ] |A: talking [0] |C: jamming [1, 0]

time 77: [ <A , <A> , <AC>, <AC>, <AC>, <AC>, <AC>, <AC>, <X> , <X> , <X> , <X> , <X> , <X> , <X> ] |A: talking [0] |C: jamming [1, 0]

time 78: [ <A , <X> , <AC>, <AC>, <AC>, <AC>, <AC>, <X> , <X> , <X> , <X> , <X> , <X> , <X> , <X> ] |A: jamming [1, 0] |C: jamming [1, 0]

W tym przykładzie węzeł A zaczął nadawać, lecz zanim sygnał doszedł do węzła C, on także zaczął transmisję. Dochodzi do kolizji – po jej wykryciu oba węzły zaczynają emitować jam signal. Widzimy jednak, że zaraz znów dojdzie do kolizji: oba węzły wylosowały czas czekania równy 0.

Przykład 2:

Po odpaleniu symulacji dla 10 milionów iteracji (prawd. 0.015, size = 15, A=1, B=9, C=15) otrzymaliśmy w podsumowaniu następujące wyniki:

SUMMARY:

Node A

Total successes: 32777

Total collisions: 29057

Node B

Total successes: 73576

Total collisions: 29142

Node C

Total successes: 44801

Total collisions: 30838

Możemy z nich zauważyć, że węzły leżące na skraju medium znacznie częściej wpadają w kolizje. Spowodowane jest to tym, że prawdopodobieństwo kolizji między dwoma węzłami jest proporcjonalne do odległości między nimi – im dalej od siebie są położone, tym większa szansa, że w trakcie podróży sygnału „a” z A do C, C zacznie nadawać.

Węzeł B, leżący po środku, najrzadziej wpada w kolizje, gdyż najszybciej dociera do pozostałych węzłów, blokując je.

Można z tego wyciągnąć wniosek, że im bliżej innych węzłów się znajdujemy, tym lepiej – szybciej wykryjemy kolizję, nie marnując czasu na nadawanie sygnału, który gdzieś jest psuty, rzadziej też będziemy w takie kolizje wpadać.

Przykład 3:

Przykład 3 jest całym wywołaniem symulacji (1000 iteracji). Znajduje się w pliku sprawozdanie3\_bonus.docx. Jest sformatowany, co ułatwia odczytywanie danych.