Technologie sieciowe, lista 3

Grupa: czwartek 17:05

Autor: Joanna Kulig

Nr indeksu: 261738

1. Opis zadania:

Zadanie zostało podzielone na dwie części:

- Napisanie programu, który umożliwia ramkowanie danych zgodnie z zasadą "rozpychania bitów" oraz odczytanie takich danych.
  - w zadaniu należało użyć metody CRC do weryfikacji poprawności ramki
  - program miał odczytywać plik tekstowy złożony z '0' i '1' (symulujący strumień bitów) i zapisywać go do ramek wraz z weryfikacją / sprawdzać poprawność ramek i przepisywać ich zawartość tak, aby otrzymać kopię pliku źródłowego
- Napisanie programu do symulowania ethernetowej metody dostępu do medium transmisyjnego CSMA/CD
  - wspólne łącze miało być realizowane za pomocą tablicy tak, żeby propagacja sygnału symulowana jest za pomoca propagacji wartości do sąsiednich komórek

Do realizacji zadania wybrałam język **Julia**.

Kod źródłowy można znaleźć na moim githubie.

2. Ramkowanie

Ramki są tworzone zgodnie z zasadą rozpychania bitów. Składają się na nią:

- flagi **01111110** informujące o początku i końcu ramki
- zawartości pliku źródłowego
- kod CRC (do jego policzenia użyłam funkcji bibliotecznej)

Rozmiar pożądanych danych w ramce został ograniczony do **32** bitów.

2.1 Rozpychanie bitów

"Rozpychanie bitów" polega na dodaniu bitu zerowego, gdy w ciągu bitów występują sekwencje mające pięć jedynek pod rząd, na przykład ciąg

```
01111111
```

po "rozpychaniu bitów" zamieni się na ciąg:

```
011111011
```

2.2 Kodowanie

Kodowanie danych do ramki:

- Czytanie danych z pliku wejściowego, odpowiadających **32 / 8** bajtom.
- Obliczamy CRC i doklejamy na koniec odczytanych bitów.
- Wykonujemy procedurę rozpychanie bitów.
- Dodajemy flagi **01111110** na początku i końcu ciągu bitów.
- Zapisujemy ramkę do pliku - jeśli nie skończyły się dane wejściowe, tworzymy koleją ramkę.

2.3 Dekodowanie

Dekodowanie danych z ramki:

- Wczytanie danych z pliku wejściowego.
- Usuwanie flag granicznych z początku i końca.
- Usuwamy dodatkowe zera, wstawione wcześniej w wyniku rozpychania bitów.
- Rozdzielamy dane na ramki.
- Dla każdej ramki:
  - Oddzielamy dane od wcześniej wyliczonego CRC.
  - Liczymy CRC dla odczytanego ciągu danych i weryfikujemy, czy jest taki sam jak poprzednio
    - jeżeli kod crc się nie zgadza lub w ramce jest błedna liczba bitów - ramka zostaje pominięta.
    - w przeciwnym przypadku zapisujemy ramkę do pliku wyjściowego.

2.4 Uruchomienie programu:

Program przyjmuje 3 argumenty:

- ścieżka pliku wejściowego
- ścieżka pliku wyjściowego
- tryb:
  - enc - kodowanie
  - dec - dekodowanie

Przykład uruchomienia:

```
julia main.jl test coded enc    # kodowanie
julia main.jl coded decoded dec # dekodowanie
```

2.5 Testy:

2.5.1 Poprawna ramka:

Plik wejściowy:

```
$ cat test
Welcome, welcome!
[...]
As always, ladies first.
[...]
Happy Hunger Games! And may the odds be ever in your favor.
```

Po kodowaniu:

```
julia main.jl test coded enc
```

```
$ cat coded
0111111001010111011001010110110001100011001011111000011111001101111000101001111110
01111110011011110110110101100101001011000100111000000101001101010100100001111110
011111100010000001110111011001010110110001001100000000000101111010111110010111110
01111110011000110110111101101101011001010010100011010001001100001001100001111110
011111100010000100001010010110110010111011010000110111011111001000110000101111110
01111110001011100010111001011101000010101100111011101011001110001011000001111110
011111100100000101110011001000000110000101101110000011111000010111100111101111110
01111110011011000111011101100001011110010000000111100101000000100010000001111110
011111100111001100101100001000000110110010101011011001111101001101101010001111110
01111110011000010110010001101001011001010000101101000011100011000111001101111110
011111100111001100100000011001100110100110101110110000011111011110010000101111110
01111110011100100111001101110100001011101010001101011100110100001000000101111110
01111110000010100101101100101110001011100110001101101001010010000011010001111110
01111110001011100101110100001010010010000110100010111001011011010000111101111110
011111100110000101110000001110000011110010000100101010001111100111000100100111110
011111100010000000100100001110101011011100100111110011100111100000000111001011111110
0111111001100111011001010111001000100000010001000111110011100111110011111011011111110
01111110010001110110000101101101011001010110110100110110110111010011001001111110
01111110011100110010000100100000001000000100110111001111000010110001111000001111110
0111111001101110011001000010000000110110100011111001111101110010010010110110111111110
011111100110000101111001001000000111010011101011101110110010000011111011101111110
0111111001101000011001010010000000110111111000111110010000000011000101010000001111110
01111110011001000110010001110011001000000010111000011101000011100010011101111110
011111100110001001100101001000000110010110111101100000001000010101111101110111111110
011111100111011001100101011100100010000001011110011011010100001001111100100111110
01111110011010010110111000100000001111001101001101000010100010110001010010111110
01111110011011110111010101110010001000000000111000011101100000001000100110111110
011111100110011001100001011101100110111110001101010000001011010001100110010111110
0111111001110010001011101110010110111000101000100100001001111110
```

Po dekodowaniu:

```
julia main.jl coded decoded dec
```

```
$ cat decoded
Welcome, welcome!
[...]
As always, ladies first.
[...]
Happy Hunger Games! And may the odds be ever in your favor.
```

2.5.2 Niepoprawna ramka:

Po zakodowaniu pliku `test` z poprzedniego przykładu, w pliku coded kilka bitów zostało zmienionych na przeciwne, a kilka zostało usuniętych.

```
$ cat coded
01111110010101110110010101101100011000110010111110000111110011011110001010011111110
01111110011011110110110101100101001011000100111000000101001101010100100001111110
01101110001000000111011101100101011011000100110000000000010111101011111001011111110
01111110011000110110111101101101011001010010100011010001001100001001100001111110
01111110001000010000101001011010010111011010000110111011111001000110000101111110
01111110001011100010111001011101000010101100111011101011001110001011000001111110
01111110010000010111001100100000011000010110111000001111100001011110011110111111110
01111110011011000111011101100001011111001000000011110010100000010001000000111110
01111110011100110011001011000010000001101100101010110110011111010011011010100011111110
01111110011000010110010001101001011001010000101101000011100011000111001101111110
01111110011100110010000011001100110100110101110110000011111011110010000101111110
01111110011100100111001101111010000101110101000110101110011010000100000101111110
01111110000010100101101100101110001011100110001101101001010010000011010001111110
01111110001011100101110100001010010010000110100010111001011011010000111101111110
01111110011000010111000001110000011110010000100101010001111100111000100100111110
01111110001000000100100001110101011011100100111110011100111100000000111001011110
01111110011001110110010101110010001000001000100011111001110011111001111010111110
01111110010001110110000101101101011001010110110100110110111010011001001111110
01111110011100110010000100100000010000010011011100111100001011000111100001111110
01111110011011100110010100100000011001011011110110000001000010101111101101111110
01111110011101100111010101110010001000001011110011011010100001001111100100111110
01111110011010010110111000100000011110011010011010000101000101100010100101111110
01111110011011110111010101110010001000000001110000111011000000100010011011111110
01111110011001100110000101110110011011111000110101000001011010001100110010111110
01111110011100100010111011100101101110001010001001000010011111010
```

Po dekodowaniu:

```
$ cat decoded
Welcome,come..]
As always, ladies first.
[...]
Happy Hunger Games! And may tdds be ein your favor.
```

Zatem program właściwie identyfikuje błędne ramki oraz je pomija.

3. Protokół CSMA / CD

3.1 Wstęp

Zadanie polegało na zasymulowaniu łącza między nadającymi węzłami. Za takie łącze miała nam posłużyć tablica, która określa przesyłane przez węzły pakiety. Została ona zrealizowana pod postacią tabicy tablic, która pokazuje, jakie pakiety znajdują się w odpowiednim segmencie.

Jednostką czasu w naszej symulacji jest iteracja, podczas której węzeł może:

- rozpocząć nadawanie
- nadawać dalej
- kończyć nadawanie
- być w stanie spoczynku

W każdej iteracji

- dane pakiety są przesuwane po łączu w odpowiednim kierunku:
  - w lewo, jeśli węzeł jest podpięty do pierwszego segmentu kabla
  - w prawo, jeśli węzeł jest podpięty do ostatniego segmentu kabla
  - w obie strony, jeśli węzeł jest podpięty do jednego ze środkowych segmentów kabla.
- urządzenie może zmienić swój stan
- może zostać wykryta **kolizja**
  - dzieje się to w momencie, gdy węzeł, który jest w trakcie nadawania, wykryje w swojej komórce pakiet, który przyszedł z innego węzła
  - w takiej sytuacji dany węzeł zatrzymuje wysyłanie danych i rozsyła po sieci pakiet, który informuje inne węzły o kolizji

Każdy pakiet musi mieć wystarczającą wielkość, aby w przypadku kolizji można było ją wykryć przed przesłaniem kolejnego pakietu, zatem możemy przyjąć tą wielkość jako dwukrotność długości naszego kabla. Musimy wtedy wykonać 2 * **długość_kabla** kroków.

W danym kroku symulacji:

- propagujemy istniejące sygnały
- sprawdzamy nadawanie z danych węzłów, tj. w razie potrzeby przerywamy nadawanie, rozpoczynamy nadawanie, kontynuujemy nadawanie, albo każemy węzłowi dalej czekać, zanim

zacznie nadawać.

3.2 Przebieg symulacji

Do symulacji potrzebujemy danych wejściowych. Zatem przyjmijmy za nasze dane wejściowe (można je łatwo zmienić w pliku simulation.jl):

```
cable_size = 10
# Węzły:
Node(name="A", position=1, idle_time=0, frames=3))
Node(name="B", position=3, idle_time=3, frames=4))
Node(name="C", position=10, idle_time=5, frames=1))
Node(name="D", position=6, idle_time=0, frames=3))
```

gdzie:

- name - nazwa węzła,
- position - określa dany segment kabla jako pozycję węzła,
- idle_time - czas oczekiwania, przed następnym nadawaniem.

Nasza sieć będzie wyglądać wtedy tak:

```
  A       B           D              C
|[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]|
```

Uruchomienie symulacji:

```
$ julia simulation.jl [mode]
```

Gdzie mode można ustawić jako "slow", dzięki czemu możemy obserwować symulacje krok po kroku, po wciśnięciu klawisza Enter.

W czasie trwania symulacji wyświetlają się następujące przebiegi:

```
Iteration: 1
A started broadcasting
B is waiting
C is waiting
D started broadcasting
Cable after 1:
[[][][][][][][][][]]

Iteration: 2
A continues broadcasting
B is waiting
C is waiting
D continues broadcasting
Cable after 2:
[[A][][][][][D][][][][]]

Iteration: 3
A continues broadcasting
B is waiting
C is waiting
D continues broadcasting
Cable after 3:
[[A][A][][][D][D][D][][][]]

Iteration: 4
A continues broadcasting
B is waiting
C is waiting
D continues broadcasting
Cable after 4:
[[A][A][A][D][D][D][D][D][][]]

Iteration: 5
A continues broadcasting
B is waiting
C is waiting
D continues broadcasting
Cable after 5:
[[A][A][A,D][A,D][D][D][D][D][D][]]

Iteration: 6
A continues broadcasting
B is waiting
C is waiting
D continues broadcasting
Cable after 6:
[[A][A,D][A,D][A,D][A,D][D][D][D][D][D]]

Iteration: 7
A detected a collision, sending collision signal
A continues broadcasting
B is waiting
C is waiting
D detected a collision, sending collision signal
```

```
D continues broadcasting
Cable after 7:
[[D,A][A,D][A,D][A,D][A,D][A,D][D][D][D][D]]

Iteration: 8
A continues broadcasting
B is waiting
C is waiting
D continues broadcasting
Cable after 8:
[[D,A][A!,D][A,D][A,D][A,D][A,D][A,D][D][D][D]]

Iteration: 9
A continues broadcasting
B is waiting
C is waiting
D continues broadcasting
Cable after 9:
[[D,A][A!,D][A!,D][A,D][A,D][A,D][A,D][A,D][D][D]]

Iteration: 10
A continues broadcasting
B is waiting
C is waiting
D continues broadcasting
Cable after 10:
[[D,A][A!,D][A!,D][A!,D][A,D][A,D][A,D][A,D][A,D][D]]

[...]
```