Sprawozdanie Lista nr 3

Przedmiot	Technologie sieciowe
Prowadzący	Mgr inż. Dominik Bojko
Autor	Maciej Bazela
Indeks	261743
Grupa	Czw. 15:15-16:55
Kod grupy	K03-76c

Kod źródłowy znajduje się w repozytorium na moim githubie.

1. Wymagania

Celem tej listy było zaimplementowanie dwóch programów:

- pierwszy z nich dotyczył ramkowania z rozpychaniem bitów,
- drugi miał symulować ethernetową metodę dostępu do medium transmisyjnego CSMA/CD.

1.1 Środowisko

Do rozwiązania tych zadań wykorzystałem język Julia.

2. Ramkowanie z rozpychaniem bitów

2.1 Struktura ramki

Ramki tworzone zgodnie z zasadą rozpychania bitów mają określoną strukturę:

```
[flaga graniczna|nagłówek|dane|crc|flaga graniczna]
```

Flagom granicznym odpowiada ciąg bitów: 01111110

Danymi, które będziemy ramkować, będą sczytane z pliku bajty.

Dla przejrzystości zakodowanych plików wynikowych przyjąłem, że zawsze sczytujemy określoną ilość bitów z pliku, np. `FRAME_SIZE=32`.

Nagłówka w tym zadaniu nie potrzebujemy, dlatego dla ułatwienia go pominąłem.

2.2 Rozpychanie bitów

"Rozpychanie" bitów polega na dodawaniu zerowego bitu po każdej pięcio-elementowej sekwencji jedynek.

Na przykład, sczytując z pliku ciąg "}}}}" i zamieniając poszczególne litery na ich wartości w UTF8:

```
01111101 01111101 01111101 01111101
```

po "rozpychaniu" otrzymamy:

```
011111001 011111001 011111001 011111001
```

2.3 CRC

Liczenie pola kontrolnego CRC pozostawiłem osobom ode mnie mądrzejszym.

Bardzo wygodny ku temu okazał się fakt, że Julia ma wbudowaną funkcję do obliczania CRC32c:

```
julia> test = "}}}}"
"}}}}"

julia> Base._crc32c(test)
0x693bb197
```

Sczytane bity z pliku wejściowego zamieniałem na String, liczyłem ich pole kontrolne CRC i zamieniałem z powrotem na bity:

```
function crc32_to_bits(str::String)::BitVector
  crc::UInt32 = Base._crc32c(str)
  return uint_to_bits(crc, 32)
end
```

2.4 Kodowanie ramek

Kodowanie ciągu znaków na ramki, polegało na:

- czytaniu `FRAME_SIZE / 8` bajtów z pliku,
- liczeniu CRC,
- "rozpychaniu" pięcio-elementowych sekwencji jedynek
- i opakowywaniu wszystkiego w ramki graniczne.

Schemat powtarzamy dopóki nie sczytaliśmy wszystkich bajtów z pliku.

Wynikowe ramki dla czytelności zamieniałem na String zer i jedynek i zapisywałem do pliku wyjściowego, podanego jako argument funkcji:

```
function encode_str(input::IO, output::IO)
  bytes = Vector{UInt8}(undef, 0)
  bits = BitVector(undef, 0)
  while !(eof(input))
    bytes = read_n_bytes(input, fld(FRAME_SIZE, 8))
    for byte in bytes
      push!(bits, uint_to_bits(byte, 8)...)
    end
    crc_bits = crc32_to_bits(bytes_to_str(bytes)) # CRC
    for crc_bit in crc_bits
      push!(bits, crc_bit)
    end
    bits_str = bits_to_str(bits)
    stuffed_str::String = replace(bits_str, r"11111" => s"111110") # Add guarding zeros
    write_str(FRAME_EDGE * stuffed_str * FRAME_EDGE * "\n", output) # Save frame to file
    empty!(bits)
  end
end
```

2.5 Dekodowanie ramek

Wczytanie zakodowanego pliku, polegało na:

- wczytaniu całego pliku jako String,
- zamianie flag granicznych (ciągów "01111110") na znak nie będący ani zerem ani jedynką (np. "|"),
- usunięciu zer powstałych przez "rozpychanie" bitów,
- rozdzielenie Stringu na osobne ramki.

Dla każdej z ramek:

- zamieniamy ciąg znaków "0" i "1" z powrotem na bity,
- rozdzielamy dane od crc (wiemy, że crc ma 32 bity, dlatego możemy bez problemu od siebie je oddzielić),
- liczymy crc dla szczytanych bitów:

- jeśli crc nie zgadza się ze sczytanym crc lub liczba bitów danych nie jest potęgą ósemki, odrzucamy ramkę,
- w przeciwnym przypadku zapisujemy zdekodowane dane do pliku wyjściowego, podanego jako argument funkcji.

```
function decode_str(input::IO, output::IO)
  input_str::String = read(input, String)
  input_str = replace(input_str, r"01111110" => s"|") # Replace frames with |
  input_str = replace(input_str, r"111110" => s"11111") # Remove guarding zeroes
  frames = split(input_str, "|")
  bits = BitVector(undef, 0)
  for frame in frames
    frame = strip(frame)
    if (isempty(frame)) continue end
    for bit in frame
      push!(bits, parse(Int, bit))
    end
    data = bits[1:end-FRAME_SIZE]
    crc = bits[end-FRAME_SIZE+1:end]
    try
      data_bytes = bits_to_bytes(data)
      if (crc != crc32_to_bits(bytes_to_str(data_bytes)))
        throw("CRC32 check failed")
      else
        for byte in data_bytes
          write_byte(byte, output)
        end
      end
    catch e
      println("Error: $e. Frame malformed, omitting.")
    end
    empty!(bits)
  end
end
```

2.6 Uruchomienie programu:

Program przyjmuje 3 argumenty:

- ścieżka pliku wejściowego
- ścieżka pliku wyjściowego
- tryb:
  - enc - kodowanie
  - dec - dekodowanie
  - chk - sprawdzenie czy dwa pliki są takie same

Przykład uruchomienia:

```
julia bit_stuffing.jl test out enc # kodowanie
julia bit_stuffing.jl out decoded dec # dekodowanie
julia bit_stuffing.jl test decoded chk # sprawdzenie
```

2.7 Przykład działania:

2.7.1 Poprawne ramki:

Plik testowy: test

```
$ cat test
Ala ma kota
Kot ma Alę
Zawsze się zastanawiałem jak to jest możliwe że kot ma Alę,
przecież to Ala jest jego właścicielką...
```

Kodowanie:

```
$ julia bit_stuffing.jl test out enc
  0.076200 seconds (316.24 k allocations: 18.363 MiB, 97.78% compilation time)
```

```
$ cat out
01111110010000010110110001100001001000000000110110101001011011000010010101111110
01111110011011010110000100100000011010110110000001100011110010101110111001111110
01111110011011110111010001100001000010101110101100100001111011001001001101111110
01111110010010110110111101110100001000001100110001001001111001000001101101111110
01111110011011010110000100100000010000010010101100111100011011000000100001111110
01111110011011001100010010011001000010100001010000010100001101000101011001111110
01111110010110100110000101110111011100110110011100011010101101101111001001111110
01111110011110100110010100100000011100111001010001011110111110000101000110111110
01111110011010011100010010011001001000000000001010001110110011101101111001111110
01111110011110100110000101110011011110100001010100111111011101001110100111001111110
01111110011000010110111001100001011101110001000000011001001010010011011011111110
01111110011000010110101100100000011101001101010001110110000011110111100001111110
01111110011011110010000001101010011001010101010000011110011100011011110101111110
01111110011100110111010000100000011011011101011010011010101001100100000101111110
01111110011011110100010110111100011011001001100000101001001110110001011101111110
01111110011010010111011101110010100100000110000001001111101111100011010100001111110
01111101100010110111100011001010010000011000000010010101101001100000101011111110
01111110011010111011101000100000011010111010110100111010101001100100000101111110
01111110011011110100010110111100011011001001100000101001001110110001011101111110
01111110011011001100010010011001001010110000010010000010000101110101100000001111110
01111110000100000000010100111000001110010110001111000011011000011111011001011111110
01111110011110100110010101100011011010010010010000010010011111001011101001011111110
01111110011001011100010110111100001000001111010111010000000110011001010001011111110
01111110011010100110010101100111011011111001001010101110101001110011000000011111110
01111110001000000111011111000010110000010101100101011001111000011001100110111111110
01111110011000011100010110011011011000011110010101010001110100010110111110011111110
01111110011010010110001101101001011001011000100111110011111000101111101110001111110
01111110011011000110101111000100100001011011010111001111100010101110101110111111110
01111110001011100010111000101110000010100100010101010101010001001010111001111110
```

Dekodowanie:

```
$ julia bit_stuffing.jl out decoded dec
  0.000262 seconds (1.49 k allocations: 93.203 KiB)
```

```
$ cat decoded
Ala ma kota
Kot ma Alę
Zawsze się zastanawiałem jak to jest możliwe że kot ma Alę,
przecież to Ala jest jego właścicielką...
```

Sprawdzenie:

```
$ julia bit_stuffing.jl test decoded chk
Before encoding chekcsum (crc32c): 276902515
After decoding chekcsum (crc32c): 276902515
Are files the same? true
```

2.7.2 Zepsute ramki:

Pozamieniam i pousuwam parę bitów z pliku "out" z poprzedniegu przykładu:

```
$ cat decoded
01111110010000010110110001100001001000000000110110101001011011000010010101111110
01111110011011010110000100100000011010110110000001100011110010101110111001111110
01111110011011110111010001100001000010101110101100100001111011001001001101111110
01111110010010110110111111101000010000011001100010010011110010000011011011111110
01111110011011010110000100100000010000010010101100111100011011000000100001111110
01111110011011001100010010011001000010100001010000010100001101000101011001111110
01111110010110100110000101110111011100110110011100011010101101101111001001111110
01111110011110100110010100100000011100111001010001011110111110000101000110111110
01111110011010011100010010011001001000000000001010001110110011101101111001111110
01111110011110100110000101110011011110100001001001111101110100111010011100111110
01111110011000010110111001100001011101110001000000011001001010010011011011111110
01111110011010010110000111000101100000100010110011000100111010111110001010111110
01111110011001010110110100100000011010101100000101100001110110000110101000111110
01111110011000010110101001000000111010011010100011101100000111101111000001111110
01111110011011110010000001001010011001010101010000011110011100011011110101111110
01111110011100110111010000100000011011011101011010011010101001100100000101111110
01111110011011110100010110111100011011001001100000101001001110110001011101111110
01100110011010010111011101100101001000001100000001001111101111100011010100001111110
01111110110001011011110001100101001000001100000001001010110100110000010101111110
01111110011010110111011110111010001000000011011011011001111000011010100001111110
01111110011011010110000100100000010000010010101100111100011011000000100001111110
01111110011011001100010010011001001010110000010010000010000101110101100000001111110
01111110001000000000101001110000011100101100011110000011011000011111011001011111110
01111110011110100110010101100011011010010010010000010010011111001011010010101111110
01111110011001010111001101111010000100000111110111010000000110011001010001011111110
01111110011010100110010101100111011011111001001010101110101001110011000000011111110
01111111000100000011101111100001011000001010110010101100111100001100110011011111110
01111110011000011100010110011011011000011110010101010001110100010110111111001111110
01111110011010010110001101101001011001011000100111110011111000101111101110001111110
01111110011011000110101111000100100001011011010111001111100010101110101110111111110
01111110001011100010111000101110000010100100010101010101010001001010111001111110
```

```
$ julia bit_stuffing.jl out decoded dec
Error: Number of bits not a power of 8. Frame malformed, omitting.
Error: Number of bits not a power of 8. Frame malformed, omitting.
Error: Number of bits not a power of 8. Frame malformed, omitting.
Error: CRC32 check failed. Frame malformed, omitting.
Error: CRC32 check failed. Frame malformed, omitting.
Error: CRC32 check failed. Frame malformed, omitting.
  0.001191 seconds (1.48 k allocations: 90.422 KiB)


$ cat decoded
Ala ma kota
ma Alę
Zawsze się anawiałem jst możlże kot ma Alę,
przecież to Ala jest jegoaścicielką...


$ julia bit_stuffing.jl test decoded chk
Before encoding chekcsum (crc32c): 276902515
After decoding chekcsum (crc32c): 1537098958
Are files the same? false
```

Jak widać, program poprawnie wyłapuje błędne ramki i je odpowiednio pomija.

3. Symulacja CSMA/CD

Do wykonania tego zadania, potrzebujemy zasymulować **łącze** pomiędzy nadającymi **urządzeniami**/**węzłami**, nadające **urządzenia**/**węzły**, **pakiety** przesyłane w sieci oraz **jednostkę czasu**, która określa co w danym momencie się dzieje w sieci.

W naszym przypadku symulowanym **łączem** będzie tablica, a dokładniej tablica tablic przesyłanych pakietów.

Każda **komórka** odpowiada położeniu danego urządzenia w sieci, tj. jeśli urządzenie podpięte jest do **komórki** 2, to urządzenie będzie "przesyłać pakiety" w tablicy na lewo i na prawo od **komórki** o indeksie 2.

Jednostką czasu w symulacji jest **krok**, a w danym kroku urządzenie może:

- spoczywać (nic nie przesyłać),
- rozpoczynać nadawanie,
- kontynuować nadawanie,
- kończyć nadawanie

W każdym kroku dany **pakiet** jest propagowany po łączu do **sąsiedniej komórki**/**komórek**, odpowiednie urządzenia są włączane/wyłączane oraz wykrywane oraz wykrywane są **kolizje**.

Kolizja następuje wtedy, kiedy urządzenie, które nadaje wykryje na swojej komórce pakiet pochodzący z innego urządzenia. W takiej sytuacji obecnie nadawany pakiet jest przerywany, a po sieci rozesłany zostaje **pakiet kolizyjny**, który informuje resztę węzłów o zaistnieniu **kolizji**.

Urządzenia nadające/**węzły** mają następujące atrybuty:

```
@kwdef mutable struct Node
  name::String = ""
  position::Int = -1
  id::Int = position
  idle::Bool = true
  idle_time::Int = -1
  collision::Bool = false
  detected_collisions::Int = 0
  frames::Int = -1
end
```

Gdzie:

- *name* - nazwa węzła,
- *position*- pozycja w tablicy,
- *id* - unikalne id urządzenia (dla prostoty *id* = *position*),
- *idle* - stan określający czy urządzenie nadaje (False), czy jest w spoczynku (True)

- *idle_time* - czas oczekiwania, przed następnym nadawaniem,
- *collision* - stan określający czy urządzenie wykryło kolizje,
- *detected_collisions* - ilość kolizji zaobserwowana przez dane urządzenie przed prawidłowym przesłaniem całego pakietu,
- *frames* - liczba pakietów/ramek przesyłana przez urządzenie, zanim przestanie w ogóle nadawać.

Pakiety określają tylko 3 wartości:

```
@kwdef mutable struct NodePacket
  node::Node = nothing
  collision_packet::Bool = node.collision
  direction::packet_directions
end
```

- *node* - węzeł/urządzenie, od którego pochodzi pakiet,
- *collision_packet* - stan określający, czy dany pakiet jest pakietem kolizyjnym,
- *direction* - w którym kierunku rozchodzi się pakiet (na lewo, na prawo, w obie strony).

Każdy pakiet musi mieć wystarczającą wielkość, aby w przypadku kolizji można było ją wykryć przed przesłaniem kolejnego pakietu.

W takim razie pakiet, będzie miał wielkość wystarczającą, aby dwukrotnie przejść przez całe łącze, przed nadaniem kolejnego pakietu.
Innymi słowy, skoro nasz kabel ma długość **n** (**n**-elementowa tablica), to pakiet musi być nadawany przez czas odpowiadający propagacji przez **2n** komórek (a skoro 1 krok == propagacja na sąsiednią komórkę, musimy wykonać **2n** kroków).

Naszą sieć symulujemy poprzez następne zmienne:

```
@kwdef mutable struct Simulation
  cable_size::Int = 0
  cable::Vector{Vector{NodePacket}} = empty_cable(cable_size)
  available_positions::Dict{Int, Node} = Dict(i => Node() for i in 1:cable_size)
  broadcasting_nodes::Vector{Node} = []
  nodes_statistics::Dict{String, Dict{Symbol, Int}} = Dict()
end
```

- *cable_size* - ilość komórek w łączu (tablicy),
- *cable* - tablica tablic pakietów, odpowiadająca temu, jakie pakiety znajdują się w danym fragmencie łącza w danym kroku,
- *available_positions* - słownik, przechowujący węzły na danych komórkach łącza,
- *broadcasting_nodes* - tablica węzłów, które musza jeszcze nadawać,
- *nodes_statistics* - statystyki dotyczące symulacji.

W danym kroku symulacji:

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

```
function step(sim::Simulation, iteration::Int)
  next_state::Vector{Vector{NodePacket}} = empty_cable(sim.cable_size)
  propagate_packets!(sim, next_state)
  broadcasts!(sim, next_state, iteration)
  sim.cable = next_state
end
```

3.1 Przebieg symulacji

W pliku main.jl znajduje się przykładowe przygotowanie i uruchomienie symulacji.

Przyjmijmy następujące atrybuty:

```
cable_size = 10
# Węzły:
Node(name="A", position=1, idle_time=0, frames=3))
Node(name="B", position=3, idle_time=5, frames=2))
Node(name="C", position=10, idle_time=10, frames=1))
Node(name="D", position=7, idle_time=0, frames=3))
Node(name="E", position=8, idle_time=0, frames=0))
```

Nasza sieć będzie wyglądać wtedy tak:

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

Uruchomienie symulacji:

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

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

Po wykonanej symulacji program zapisuje jej przebieg do pliku nazwazmiennej_out.log:

```
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][A,D][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][A,D][A,D][A,D][A,D][D][D][D][D][D]|

Iteration: 7
A continues broadcasting
B is waiting
C is waiting
D continues broadcasting
Cable after 7:
|[A][A,D][A,D][A,D][A,D][A,D][A,D][D][D][D][D]|

Iteration: 8
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 8:
|[D,A][A,D][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][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]|

[...]
```

Aby zmienić atrybuty symulacji, wystarczy zmienić kod w main.jl.

Funkcja:

- Simulation(cable_size) - tworzy symulacje o danej wielkości łącza,
- new_node(name, position, idle_time, frames) - tworzy nowy node,
- add_node!(simulation, node) - dodaje node do symulacji,
- run(simulation, slow=mode) - uruchamia symulacje w danym mode ("slow",albo domyślnie bez przerywania),
- statistics(simulation) - drukuje statystyki węzłów po symulacji.

Wszystkie funkcje, potrzebne do wykonania symulacji, przechowywane są w module `CSMA_CD_Simulation` w pliku *simulation.jl*, a struktury węzła i pakietu w pliku *node*.jl.

Moduł `CSMA_CD_Simulation` eksportuje także wektor `messages` zawierający logi po wykonaniu symulacji.