Lista 3

Paweł Kajanek

1 Framer.py

Moduł 'Framer' zawiera zbiór funcji, których można użyć do symulacji ramkowania pakietów. Dwiema głównymi funkcjami są *pack*() i *unpack*(). Pierwsza z nich wykonuje kolejne operacje:

1. Obliczenie CRC dla wejściowego łańcucha znaków.
2. Konkatenacja wejściowego łańcucha z obliczonym CRC.
3. 'Rozepchnięcie bitów' w tak powstałym łańcuchu.
4. Dodanie ciągów '01111110' na początku i końcu łańcucha jako nagłówek i pole końca ramki.

Druga funkcja natomiast wykonuje operacje odwrotne w odwrotnej kolejności:

1. Usunięcie ciągów '01111110' na początku i końcu łańcucha wejściowego.
2. 'Zepchnięcie bitów' w tak powstałym łańcuchu.
3. Oddzielenie CRC od wiadomości.
4. Sprawdzenie czy CRC otrzymanej wiadomości jest równe CRC w nadesłanym pakiecie.

W przypadku, gdy CRC jest poprawe, *unpack*() zwraca odkodowaną wiadomość, w przeciwnym przypadku - 0. Pozostałe funcje służą do zapisywania pakietów w plikach, odczytywania zawartości plików i tworzeniu losowych ciągów znaków.

2 Medium.py

Program zawiera klasę 'Medium', która symuluje działanie jednowymiarowego medium rozgłoszeniowego. Klasa zawiera 2 tablice pomocnicze. Po użyciu metody *propagate*() elementy w jednej z nich są przesuwane w prawo, a w drugiej w lewo. Obie tablice są dopełniane zerami. Metoda *display*() drukuje sumy elmentów tych tablic na odpowiadających indeksach ($t_1[i] + t_2[i]$ dla wszytkich i). Metoda *send*() dodaje podany bit w podany punkt medium, a metoda *get_bit*() zwraca sumę elmentów tablic pomocniczych w podanym miejscu ($t_1[i] + t_2[i]$ da danego i).

3 Device.py

Program zawiera klasę 'Device', która symuluje działanie urządzenia podłączonego do medium (wysyłanie wiadomości i rozwiązywanie kolizji). Klasa zawiera kolejkę wiadomości do której można je dodawać za pomocą metody *add_message*(). Metody *work*() i *resolve_collision*() reprezentują procedury CSMA/CD. Wyglądają one następująco:
work():

1. Jeśli urządzenie powinno czekać, to niech nic nie robi w tej turze.
2. Jeśli nie ma aktualnie wiadomości w buforze, to załaduj jedną wiadomość z kolejki do bufora.
3. Jeśli medium nie jest wolne, to sprawdź czy nie wystąpiła kolizja.
4. Jeśli medium jest wolne, to wyślij kolejny bit wiadomość.

resolve_collision() (wywoływana jeśli zostanie wykryta kolizja):

1. Wyślij 'jam signal'.
2. Zwiększ licznik prób retransmisji a o 1.
3. Jeśli przekroczono maksymalną ilośc prób, zgłoś informację, że wysłanie wiedomości nie powiodło się.
4. W przeciwnym przypadku wylosuj liczbę $r \in \{1, 2, \ldots, 2^a - 1\}$.
5. Użyj metody $wait(r \cdot medium.length)$ (w zasadzie, powinno się mnożyć r przez maksymalną długość medium, jednak w tym przypadku nie jest to konieczne).
6. Powróć do normalnego trybu pracy.

Metoda *wait*() spowoduje, że przez następne $r \cdot medium.length$ cykli (wywołań *work*()) urządzenie będzie bezczynne.
Pozostałe metody klasy Device są metodami pomocniczymi, które sprawiają, że kod w wyżej opisanych metodach jest czytelniejszy.

4 Przykładowe programy

4.1 Framer.py

W programie 'Framer.py' znajduje się również przykładowa funkcja *main*(), która zostanie wywołana jeśli uruchomimy ten program. Stworzone wtedy zostaną 3 pliki:

- 'Z.txt' zawierający losowe znaki,
- 'W.txt' zawierający utworzone z poprzedniego pliku pakiety (złożone z '0' i '1'),
- 'Z2.txt' zawierający odtworzone z pakietów wiadomości (powinien być identyczny z plikiem 'Z.txt').

4.2 Example.py

Program korzystając z modułów device.py i medium.py tworzy medium długości 100 oraz 2 urządzenia podłączone do medium w punktach 4 i 70. Następnie do obu urządzeń dodane zostają 3 wiadomości składające się ze 100 znaków '1' i uruchomiona zostaje główna pętla programu, w której symulowana jest praca urządzeń i propagacja bitów w medium. Po każdej pętli wyświetlony zostaje stan medium (*medium.display*()). Pętla wykonuje się 1000 razy.