Obraz zawierający tekst, Czcionka, logo, zrzut ekranu

Opis wygenerowany automatycznie

**Wydział Informatyki i Telekomunikacji**

Kierunek studiów: **Informatyka techniczna**

Prowadzący: **dr hab. inż. Stanisław Piestrak, prof. PWr**

Termin zajęć: **czwartek NP. 9:15-11:00**

Data oddania projektu: **- 6.06.2024**

Niezawodność i Diagnostyka Układów Cyfrowych

Transmisja w systemie ARQ

Wojciech Kozioł (272947), Cyprian Kozubek (272959)

Spis treści

[2 Cele i opis projektu 3](#_Toc168235023)

[3 Transmisja ARQ 3](#_Toc168235024)

[3.1 Warianty ARQ 3](#_Toc168235025)

[3.2 Stop-and-Wait ARQ 3](#_Toc168235026)

[4 Modele transmisji 4](#_Toc168235027)

[4.1 Model BSC 4](#_Toc168235028)

[4.1.1 Charakterystyka modelu 4](#_Toc168235029)

[4.1.2 Prawdopodobieństwo błędu 5](#_Toc168235030)

[4.2 Model Gilberta-Eliota 5](#_Toc168235031)

[4.2.1 Charakterystyka modelu 5](#_Toc168235032)

[5 Kody detekcyjne 5](#_Toc168235033)

[5.1 Bit parzystości 5](#_Toc168235034)

[5.1.1 Działanie bitu parzystości 5](#_Toc168235035)

[5.2 CRC8, CRC16, CRC32 6](#_Toc168235036)

[5.2.1 Działanie CRC 6](#_Toc168235037)

[6 Badania 6](#_Toc168235038)

6.1. Opis przeprowadzonych badań…………………………………………………………….6

6.2. Wyniki eksperymentu………………………………………………………………………….7

6.2.1. Model BSC…………………………………………………………………………………………………7

6.2.2. Model Gilberta-Elliotta………………………………………………………………………………13

6.2.3. Omówienie oraz wnioski dotyczące całości badań…………………………………………20

**7 Opis kodu programu…………………………………………………………………………22**

7.1. Klasa main.py……………………………………………………………………………………………..22

7.2. Klasa Endcoder.py……………………………………………………………………………………….23

7.3. Klasa Decoder.py…………………………………………………………………………………….…..24

7.4. Klasa receiver.py………………………………………………………………………………………….26

7.5. Klasa sender.py……………………………………………………………………………………………28

[8 Opis programu 33](#_Toc168235039)

[8.1 Użyte biblioteki zewnętrzne 33](#_Toc168235040)

[9 Bibliografia 33](#_Toc168235041)

# Cele i opis projektu

Głównym celem projektu jest zaprojektowanie i zaimplementowanie symulacji mechanizmu ARQ (Automatic Repeat Request), który wykorzystuje kody detekcyjne błędów do poprawy niezawodności i efektywności transmisji danych w systemach komunikacyjnych. Projekt ma na celu zilustrowanie jak techniki ARQ mogą minimalizować wpływ błędów transmisji na jakość przesyłanych danych, co jest kluczowe w aplikacjach wymagających wysokiej niezawodności, takich jak komunikacja satelitarna, telekomunikacja czy sieci komputerowe. Projekt składa się z implementacji symulatora ARQ, enkodera i dekodera.

# Transmisja ARQ

Transmisja ARQ to technika stosowana w komunikacji cyfrowej, mająca na celu zapewnienie poprawnej dostawy danych poprzez automatyczne wykrywanie błędów i żądanie ponownego przesłania danych, które zostały zakłócone [1]. Jest to podstawowy mechanizm w wielu systemach komunikacji do zwiększenia niezawodności przesyłu danych.

## Warianty ARQ

Istnieją różne warianty mechanizmu ARQ, które różnią się sposobem zarządzania przepływem danych i reagowaniem na błędy [2]:

* **Stop-and-Wait ARQ**: Nadawca czeka na potwierdzenie każdego wysłanego pakietu przed wysłaniem kolejnego. Ten wariant transmisji ARQ został zaimplementowany w projekcie. Minusem tego rozwiązania może być długi czas oczekiwania potwierdzenia odebrania poprawnego pakietu.
* **Go-Bach-N ARQ**: Nadawca może wysłać wiele pakietów bez oczekiwania na potwierdzenie, ale w przypadku wykrycia błędu musi powtórzyć wysyłanie wszystkich pakietów począwszy od pierwszego błędnie odebranego. Oczywistym minusem tego rozwiązania jest ilość pakietów, jakie trzeba wysłać ponownie po otrzymaniu wiadomości o błędzie, który wystąpił.
* **Selective Repeat ARQ**: W tym wariancie, tylko pakiet z błędem jest ponownie przesyłany, a odbiorca jest w stanie przechować pakiety odebrane poprawnie, które nadeszły po błędnie przesłanym pakiecie. Zwiększa efektywność, szczególnie w kanałach z dużą przepustowością i dużym opóźnieniem, jednak odbiorca potrzebuje znacznie więcej pamięci na przechowanie wszystkich pakietów.

## Stop-and-Wait ARQ

Transmisja ARQ w wariancje Stop-and-Wait, która została zaimplementowana w projekcie, charakteryzuje się oczekiwaniem przez system wysyłający pakiety potwierdzenia przyjęcia przez odbiornik pakietu z poprawnymi danymi z użyciem kodów detekcyjnych [3]. W przypadku, gdy nadawca nie otrzyma takiego potwierdzenia, wysyła pakiet ponownie. Dużym plusem tego wariantu jest to, że odbiorca nie potrzebuje żadnej dodatkowej pamięci na przechowywanie odebranych pakietów oraz fakt, że żaden z pakietów nie jest wysyłany bez potrzeby. Minusem tego rozwiązania jest długi czas oczekiwania na potwierdzenie, który mógłby być spożytkowany np. na wysyłanie kolejnego pakietu.

Obraz zawierający tekst, diagram, linia, zrzut ekranu

Opis wygenerowany automatycznie

Rysunek 1: Schemat transmisji Stop-and-Wait ARQ [3].

# Modele transmisji

## Model BSC

Model Binary Symmetric Channel (BSC) to matematyczny model kanału transmisyjnego używany do opisu kanału, który może przesyłać jedynie dwa symbole (najczęściej bit 0 i bit 1) i jest podatny na błędy w symetryczny sposób [4]. Kanał ten jest kluczowym narzędziem do analizy systemów transmisyjnych, które mogą doświadczać błędów w postaci przekłamań bitów.

### Charakterystyka modelu

* **Dwa stany wejściowe i wyjściowe**: BSC może przyjmować i przekazywać dwa różne symbole, zazwyczaj reprezentowane jako 0 i 1.
* **Błąd symetryczny**: Prawdopodobieństwo przekłamania bitu jest takie samo dla obu stanów. Oznacza to, że prawdopodobieństwo zmiany 0 na 1 jest równe prawdopodobieństwu zmiany 1 na 0.
* **Niepamięciowość**: BSC jest kanałem niepamięciowym, co oznacza, że przekłamanie jednego bitu jest niezależne od przekłamań innych bitów przesyłanych przez kanał.

### Prawdopodobieństwo błędu

W modelu BSC, błąd transmisji jest definiowany przez stałą wartość prawdopodobieństwa przekłamania , gdzie . Prawdopodobieństwo, że bit zostanie przekazany poprawnie (niezmieniony), jest . Na przykład, jeśli , to oznacza, że średnio jeden na sto bitów zostanie błędnie przekazany.

## Model Gilberta-Elliotta

Model Gilberta-Eliota to bardziej złożony model kanału transmisyjnego, który poszerza ideę prostego BSC poprzez wprowadzenie koncepcji dwóch stanów kanału: dobrego i złego [6]. W przeciwieństwie do BSC, w którym prawdopodobieństwo błędu jest stałe, model Gilberta-Elliotta uwzględnia zmienność tego prawdopodobieństwa w czasie, co pozwala na bardziej realistyczne odwzorowanie fizycznych kanałów transmisyjnych, w których warunki mogą się zmieniać.

### Charakterystyka modelu

* **Dwa stany kanału**
  + Stan dobry (G): Kanał w tym stanie charakteryzuje się niskim prawdopodobieństwem błędu, co oznacza, że bity są przesyłane z dużą dokładnością.
  + Stan zły (B): W tym stanie prawdopodobieństwo błędu jest znacznie wyższe, co prowadzi do częstszego przekłamywania bitów.
* **Przejścia między stanami:** Model obejmuje prawdopodobieństwa przejść między stanami, które są modelowane jako proces Markowa [6]. Prawdopodobieństwo przejścia ze stanu dobrego do złego oraz ze stanu złego do dobrego wpływa na dynamikę jakości transmisji.
* **Niepamięciowość**: Tak jak w BSC, Gilbert-Elliott jest modelem niepamięciowym w obrębie danego stanu, co oznacza, że błędy wewnątrz jednego stanu są niezależne od siebie.

# Kody detekcyjne

## Bit parzystości

Kontrola parzystości polega na dodaniu do wysyłanej wiadomości jednego dodatkowego bitu kontrolnego, który pozwala na wykrycie pojedynczych błędów bitowych w każdym zestawie bitów [7].

### Działanie bitu parzystości

1. **Obliczanie parzystości**

Bit parzystości ustawiany jest tak, żeby całkowita liczba jedynek w słowie danych była parzysta.

1. **Przykład**

Zakładając, że dane wyglądają następująco: 10011010, liczba jedynek w słowie wynosi 4, co jest liczbą parzystą. W tym przypadku bit parzystości wynosi 0 i jest on dodawany na koniec ciągu danych.

1. **Weryfikacja**

Odbiorca, otrzymując dane, oblicza parzystość danego słowa włączając otrzymany bit parzystości. Jeżeli liczba jedynek jest parzysta, odbiorca zakłada, że dane nie zawierają błędów. Jeżeli jest nieparzysta, oznacza to, że wystąpił błąd.

## CRC8, CRC16, CRC32

CRC8, czyli 8-bitowy kod cyklicznej redundancji (Cyclic Redundancy Check), jest popularną metodą detekcji błędów w transmisji danych, szczególnie w aplikacjach o niskiej mocy i prostych komunikacjach cyfrowych. CRC jest używany do wykrywania przypadkowych zmian w surowych danych przesyłanych w sieciach lub przechowywanych na nośnikach danych. Jest szczególnie efektywny w wykrywaniu błędów spowodowanych przez zakłócenia w kanałach transmisyjnych.

### Działanie CRC

1. **Definicja wielomianu**

CRC8 wykorzystuje zdefiniowany wielomian, który jest zwykle ustalony dla danego standardu lub aplikacji. Na przykład, popularny wielomian dla CRC-8 to . Wielomian ten określa sposób, w jaki dane są przetwarzane podczas obliczeń CRC.

1. **Przygotowanie danych**

Aby obliczyć CRC, do oryginalnych danych dodaje się 8 zer na końcu. Długość dopisanych zer odpowiada długości CRC (w tym przypadku 8 bitów).

1. **Dzielenie modulo 2**

Następnie, powiększone dane są dzielone przez wielomian CRC przy użyciu operacji dzielenia modulo 2 (bez przeniesienia). Proces ten jest podobny do dzielenia binarnego, ale bez dodawania.

1. **Obliczanie reszty**

Wynik dzielenia, czyli reszta, to wartość CRC. Ma ona 8 bitów i jest dołączana do oryginalnych danych jako suma kontrolna przed ich wysłaniem.

1. **Weryfikacja**

Odbiorca, otrzymując dane z dołączonym CRC, wykonuje te same obliczenia na całym ciągu (dane plus otrzymany CRC). Jeżeli transmisja była bezbłędna, wynik końcowy dzielenia (nowa reszta) powinien wynieść zero, co oznacza, że dane są prawidłowe. W przeciwnym razie, jakiekolwiek niezerowe wyniki wskazują na wystąpienie błędu.

Dla przykładu dokładnie został opisany kod CRC8. Kody CRC16 i CRC32 działają analogicznie.

# Badania

## Opis przeprowadzonych badań

Przeprowadzono badania symulacyjne skuteczności transmisji dla różnych parametrów kanału, wyznaczono BER(Bit Error Rate), błędy niezależne oraz błędy grupowe dla transmisji w systemie ARQ oraz bez użycia tego systemu, użyto 4 rodzajów kodów detekcyjnych: bitu parzystości, CRC8, CRC16 oraz CRC32. Badania wykonano dla losowo generowanego ciągu 2048 bitów dla modeli BSC oraz Gilberta-Elliotta dla dwóch prawdopodobieństw wystąpienia błędu transmisji oraz , dla prawdopobieństwa G/B oraz B/G w systemie Gilberta-Elliotta wynoszącej 0.1, dla liczby dopuszczalnych błędnych ramek 0, 10, 15, 20, 50 oraz 100. Błędy grupowe oraz błędy niezależne w systemie ARQ były całkowicie niwelowane w przeprowadzonych badaniach, dlatego w tabelach oraz wykresach umieszczono tylko błędy grupowe oraz niezależne w transmisji bez systemu ARQ.

## Wyniki eksperymentu

### Model BSC

1. Bit parzystości

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bit parzystości - Prawdopodobieństwo błędu | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0,0001 | 0,0023 | 0,0064 |
| Bez ARQ | BER | 0,488336 | 0,4951 | 0,4995 | 0,500977 | 0,5256 | 0,583398 |
| Bez ARQ | błędy grupowe | 478 | 486 | 487 | 498 | 513 | 574 |
| Bez ARQ | błędy niezależne | 1012 | 1027 | 1034 | 1047 | 1065 | 1099 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bit parzystości - Prawdopodobieństwo błędu | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0 | 0,0001 | 0,0025 |
| Bez ARQ | BER | 0,443258 | 0,449158 | 0,45048 | 0,46098 | 0,472122 | 0,50893 |
| Bez ARQ | błędy grupowe | 471 | 478 | 483 | 488 | 501 | 534 |
| Bez ARQ | błędy niezależne | 1002 | 1014 | 1023 | 1026 | 1045 | 1095 |

1. CRC8

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CRC8 - Prawdopodobieństwo błędu | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0,0001 | 0,0011 | 0,0027 |
| Bez ARQ | BER | 0,483023 | 0,4838 | 0,4936 | 0,496469 | 0,515766 | 0,580875 |
| Bez ARQ | błędy grupowe | 468 | 476 | 483 | 487 | 497 | 560 |
| Bez ARQ | błędy niezależne | 1008 | 1017 | 1020 | 1033 | 1054 | 1087 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| CRC8 - Prawdopodobieństwo błędu | | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0 | 0 | 0,0009 |
| Bez ARQ | BER | 0,432188 | 0,43823 | 0,445002 | 0,45023 | 0,46799 | 0,49332 |
| Bez ARQ | błędy grupowe | 463 | 470 | 478 | 485 | 491 | 512 |
| Bez ARQ | błędy niezależne | 991 | 998 | 1011 | 1013 | 1036 | 1072 |

1. CRC 16

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CRC16 - Prawdopodobieństwo błędu | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0 | 0,0007 | 0,0017 |
| Bez ARQ | BER | 0,473492 | 0,479 | 0,4863 | 0,489258 | 0,5091 | 0,554 |
| Bez ARQ | błędy grupowe | 469 | 475 | 481 | 484 | 491 | 547 |
| Bez ARQ | błędy niezależne | 1006 | 1013 | 1017 | 1028 | 1046 | 1070 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CRC16 - Prawdopodobieństwo błędu | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0 | 0 | 0,0001 |
| Bez ARQ | BER | 0,429121 | 0,429658 | 0,432894 | 0,439976 | 0,460977 | 0,48443 |
| Bez ARQ | błędy grupowe | 468 | 470 | 476 | 482 | 486 | 504 |
| Bez ARQ | błędy niezależne | 981 | 990 | 996 | 1002 | 1022 | 1056 |

1. CRC32

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CRC32 - Prawdopodobieństwo błędu | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0 | 0 | 0,0002 |
| Bez ARQ | BER | 0,4723 | 0,478 | 0,4829 | 0,486328 | 0,49874 | 0,5289 |
| Bez ARQ | błędy grupowe | 464 | 468 | 472 | 476 | 484 | 502 |
| Bez ARQ | błędy niezależne | 979 | 992 | 1002 | 1014 | 1023 | 1049 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CRC32 - Prawdopodobieństwo błędu | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0 | 0 | 0 |
| Bez ARQ | BER | 0,428438 | 0,428812 | 0,429377 | 0,431721 | 0,450022 | 0,471922 |
| Bez ARQ | błędy grupowe | 459 | 464 | 468 | 471 | 479 | 497 |
| Bez ARQ | błędy niezależne | 953 | 986 | 989 | 996 | 1009 | 1038 |

1. Podsumowanie oraz wnioski

**Wpływ prawdopodobieństwa błędu**:

Przy wyższym prawdopodobieństwie błędu (), wartości BER są generalnie wyższe niż przy niższym ().

Wzrasta także liczba błędów grupowych i niezależnych przy wyższym prawdopodobieństwie błędu.

**Efektywność ARQ**:

W systemie z ARQ, BER, liczba błędów grupowych i niezależnych wynosi 0, co wskazuje na skuteczne wykrywanie i korekcję błędów. Dopiero przy bardzo dużej liczby dopuszczalnych błędnych ramek, zauważyć można nieznaczny wzrost BER, niwelowany coraz skuteczniej z użyciem coraz lepszego kodu korekcyjnego.

**Porównanie metod korekcji**:

Metody CRC (CRC8, CRC16, CRC32) generalnie wykazują lepsze wyniki w porównaniu do bitu parzystości, co jest widoczne w niższych wartościach BER oraz mniejszej liczbie błędów grupowych i niezależnych. Zauważono także, że każdy kolejny kod CRC osiągał lepsze wyniki od poprzedniego, lepiej niwelując BER i błędy grupowe oraz niezależne.

**Tendencje wzrostowe**:

Wraz ze wzrostem liczby dopuszczalnych błędów w ramce, BER, liczba błędów grupowych oraz liczba błędów niezależnych wzrasta dla wszystkich metod korekcji błędów, zauważalny jest także wzrost BER.

Podsumowując, dane wskazują na znaczący wpływ prawdopodobieństwa błędu transmisji na wyniki BER oraz liczby błędów grupowych i niezależnych. System ARQ skutecznie eliminuje błędy i poprawia skuteczność transmisji, a metody CRC są bardziej efektywne niż bity parzystości.

### Model Gilberta-Elliotta

1. Bit parzystości

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bit parzystości - Prawdopodobieństwo błędu | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0,0022 | 0,0045 | 0,0098 |
| Bez ARQ | BER | 0,47998 | 0,488535 | 0,498047 | 0,500977 | 0,522949 | 0,554321 |
| Bez ARQ | błędy grupowe | 469 | 478 | 488 | 495 | 509 | 537 |
| Bez ARQ | błędy niezależne | 1012 | 1027 | 1034 | 1047 | 1071 | 1101 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bit parzystości - Prawdopodobieństwo błędu | | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0,0007 | 0,00021 | 0,0066 |
| Bez ARQ | BER | 0,440578 | 0,454158 | 0,461548 | 0,467098 | 0,474122 | 0,495093 |
| Bez ARQ | błędy grupowe | 465 | 473 | 483 | 488 | 494 | 524 |
| Bez ARQ | błędy niezależne | 983 | 1011 | 1020 | 1026 | 1065 | 1095 |

1. CRC8

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CRC8 - Prawdopodobieństwo błędu | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0,0003 | 0,0033 | 0,0078 |
| Bez ARQ | BER | 0,472342 | 0,483887 | 0,491699 | 0,495605 | 0,507324 | 0,522638 |
| Bez ARQ | błędy grupowe | 468 | 474 | 483 | 489 | 498 | 524 |
| Bez ARQ | błędy niezależne | 1008 | 1017 | 1020 | 1033 | 1054 | 1087 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CRC8 - Prawdopodobieństwo błędu | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0 | 0,0005 | 0,0027 |
| Bez ARQ | BER | 0,437219 | 0,442305 | 0,452002 | 0,463023 | 0,48013 | 0,504332 |
| Bez ARQ | błędy grupowe | 463 | 468 | 477 | 484 | 491 | 518 |
| Bez ARQ | błędy niezależne | 980 | 991 | 1015 | 1022 | 1039 | 1072 |

1. CRC16

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CRC16 - Prawdopodobieństwo błędu | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0 | 0,0019 | 0,0054 |
| Bez ARQ | BER | 0,471021 | 0,481234 | 0,489923 | 0,49377 | 0,49982 | 0,513992 |
| Bez ARQ | błędy grupowe | 463 | 466 | 473 | 479 | 494 | 515 |
| Bez ARQ | błędy niezależne | 981 | 987 | 992 | 1005 | 1023 | 1071 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CRC16 - Prawdopodobieństwo błędu | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0 | 0,0001 | 0,0012 |
| Bez ARQ | BER | 0,433121 | 0,438658 | 0,449375 | 0,451458 | 0,470108 | 0,498664 |
| Bez ARQ | błędy grupowe | 458 | 462 | 467 | 474 | 487 | 506 |
| Bez ARQ | błędy niezależne | 977 | 985 | 997 | 1008 | 1017 | 1069 |

1. CRC32

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CRC32 - Prawdopodobieństwo błędu | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0 | 0 | 0,0007 |
| Bez ARQ | BER | 0,469876 | 0,476772 | 0,483929 | 0,489221 | 0,493221 | 0,50772 |
| Bez ARQ | błędy grupowe | 457 | 462 | 467 | 472 | 489 | 505 |
| Bez ARQ | błędy niezależne | 966 | 972 | 983 | 991 | 1000 | 1034 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CRC32 - Prawdopodobieństwo błędu | | | | | | | |
|  | Ilość błędnych ramek | 0 | 10 | 15 | 20 | 50 | 100 |
| ARQ | BER | 0 | 0 | 0 | 0 | 0 | 0,0001 |
| Bez ARQ | BER | 0,426738 | 0,431117 | 0,437938 | 0,441721 | 0,4625 | 0,491922 |
| Bez ARQ | błędy grupowe | 450 | 457 | 462 | 468 | 485 | 500 |
| Bez ARQ | błędy niezależne | 963 | 969 | 977 | 985 | 995 | 1024 |
|  |  |  |  |  |  |  |  |

1. Podsumowanie oraz wnioski

**Wpływ prawdopodobieństwa błędu**:

* + Przy wyższym prawdopodobieństwie błędu (), wartości BER są generalnie wyższe niż przy niższym ().
  + Wzrasta także liczba błędów grupowych i niezależnych przy wyższym prawdopodobieństwie błędu.

**Efektywność ARQ**:

* + W systemie z ARQ, BER wynosi 0 dla większości badań, co wskazuje na skuteczne wykrywanie i korekcję błędów. Wartości BER po znacznym wzroście dopuszczalnych błędnych ramek osiągały małe wartości. System ARQ redukował błędy grupowe i niezależne do zera.

**Porównanie metod korekcji**:

* + Metody CRC (CRC8, CRC16, CRC32) generalnie wykazują lepsze wyniki w porównaniu do bitu parzystości, co jest widoczne w niższych wartościach BER oraz mniejszej liczbie błędów grupowych i niezależnych. Także coraz lepszy kod korekcyjnych osiągał coraz lepsze wyniki.

**Tendencje wzrostowe**:

* + Wraz ze wzrostem liczby błędów w ramce, BER, liczba błędów grupowych oraz liczba błędów niezależnych wzrasta dla wszystkich metod korekcji błędów.

Podsumowując, dane wskazują na znaczący wpływ prawdopodobieństwa błędu transmisji na wyniki BER oraz liczby błędów grupowych i niezależnych. System ARQ skutecznie eliminuje błędy, a metody CRC są bardziej efektywne niż bity parzystości. Model Gilberta-Elliotta pokazuje, że zarówno BER, jak i liczba błędów rośnie wraz ze wzrostem prawdopodobieństwa błędu, co jest zgodne z oczekiwaniami.

### Omówienie oraz wnioski dotyczące całości badań

#### Ogólne obserwacje:

* **Model BSC (Binary Symmetric Channel)** oraz **model Gilberta-Elliotta** prezentują różne wyniki BER, liczby błędów grupowych i liczby błędów niezależnych w zależności od zastosowanej metody korekcji błędów i prawdopodobieństwa błędu transmisji.
* Zarówno w modelu BSC, jak i w modelu Gilberta-Elliotta, system ARQ skutecznie eliminuje błędy (BER = 0).

6.2.3.2. Analiza szczegółowa:

#### Prawdopodobieństwo błędu transmisji

* **Bit parzystości**:
  + **BSC**: BER bez ARQ wzrasta z 0,488 do 0,583.
  + **Gilberta-Elliotta**: BER bez ARQ wzrasta z 0,480 do 0,554.
  + Błędy grupowe i niezależne w modelu Gilberta-Elliotta są nieco wyższe niż w BSC.
* **CRC8**:
  + **BSC**: BER bez ARQ wzrasta z 0,483 do 0,580.
  + **Gilberta-Elliotta**: BER bez ARQ wzrasta z 0,472 do 0,523.
  + Model Gilberta-Elliotta ma nieco niższe wartości BER przy wyższym prawdopodobieństwie błędu.
* **CRC16**:
  + **BSC**: BER bez ARQ wzrasta z 0,473 do 0,554.
  + **Gilberta-Elliotta**: BER bez ARQ wzrasta z 0,471 do 0,514.
  + Model Gilberta-Elliotta ponownie wykazuje nieco niższe wartości BER.
* **CRC32**:
  + **BSC**: BER bez ARQ wzrasta z 0,472 do 0,528.
  + **Gilberta-Elliotta**: BER bez ARQ wzrasta z 0,470 do 0,508.
  + Podobnie jak w poprzednich przypadkach, model Gilberta-Elliotta pokazuje niższe wartości BER.

#### Prawdopodobieństwo błędu transmisji

* **Bit parzystości**:
  + **BSC**: BER bez ARQ wzrasta z 0,443 do 0,508.
  + **Gilberta-Elliotta**: BER bez ARQ wzrasta z 0,441 do 0,495.
  + Model Gilberta-Elliotta ma niższe wartości BER i błędów grupowych w porównaniu do BSC.
* **CRC8**:
  + **BSC**: BER bez ARQ wzrasta z 0,432 do 0,493.
  + **Gilberta-Elliotta**: BER bez ARQ wzrasta z 0,437 do 0,504.
  + W tym przypadku model BSC pokazuje niższe wartości BER w porównaniu do modelu Gilberta-Elliotta.
* **CRC16**:
  + **BSC**: BER bez ARQ wzrasta z 0,429 do 0,484.
  + **Gilberta-Elliotta**: BER bez ARQ wzrasta z 0,433 do 0,499.
  + Podobnie jak w poprzednich przypadkach, model BSC wykazuje niższe wartości BER.
* **CRC32**:
  + **BSC**: BER bez ARQ wzrasta z 0,428 do 0,471.
  + **Gilberta-Elliotta**: BER bez ARQ wzrasta z 0,427 do 0,492.
  + Model BSC ma nieco niższe wartości BER.

### Wnioski:

**Porównanie modeli**:

* + **Model BSC** generalnie wykazuje nieco wyższe wartości BER przy wyższym prawdopodobieństwie błędu transmisji (), ale niższe przy niższym ()w porównaniu do modelu Gilberta-Elliotta.
  + **Model Gilberta-Elliotta** pokazuje większą zmienność wyników BER w zależności od liczby błędów, co jest charakterystyczne dla tego modelu ze względu na jego naturę (błędy występują w grupach).

**Efektywność metod korekcji błędów**:

* + Wszystkie metody korekcji błędów (Bit parzystości, CRC8, CRC16, CRC32) są skuteczne w obydwu modelach, z CRC16 i CRC32 pokazującymi lepsze wyniki w obu przypadkach.
  + **CRC16 i CRC32** są bardziej efektywne w redukcji BER w porównaniu do Bitu parzystości i CRC8, zarówno w modelu BSC, jak i Gilberta-Elliotta.

**Tendencje wzrostowe**:

* + W obu modelach wzrost liczby błędów w ramce skutkuje wzrostem BER, liczby błędów grupowych oraz liczby błędów niezależnych.
  + Model Gilberta-Elliotta ma tendencję do wyższej liczby błędów grupowych w porównaniu do modelu BSC, co jest zgodne z jego naturą.

Podsumowując, choć system ARQ skutecznie eliminuje błędy w obu modelach, różnice w wartości BER i liczbie błędów wskazują na specyficzne cechy każdego z modeli transmisji, z modelami CRC (szczególnie CRC16 i CRC32) pokazującymi lepsze wyniki w redukcji błędów. Dla większości kodów korekcyjnych model BSC pokazywał lepsze wyniki BER niż model Gilberta-Elliotta, tylko bit parzystości. W modelu BSC zauważono także mniejszą ilość błędów grupowych   
 i niezależnych aniżeli w modelu Gilberta-Elliotta. Badania dowiodły również skuteczności zaimplementowanego systemu, gdyż zgodnie z założeniami teoretycznymi transmisja w systemie ARQ lepiej niwelowała błędy transmisji   
 i korygowała wszystkie błędy.

# Opis kodu programu

* 1. Klasa main.py

Ten kod tworzy prosty serwer TCP, w którym przeprowadzane są badania działania systemu ARQ, który nasłuchuje połączeń na lokalnym adresie IP 127.0.0.1 i porcie 12345. Opis krokowy co robi kod:

1. Importuje moduł socket, który jest niezbędny do komunikacji sieciowej.
2. Definiuje funkcję start\_server().
3. Wewnątrz tej funkcji tworzy gniazdo serwera za pomocą socket.socket(socket.AF\_INET, socket.SOCK\_STREAM).
4. Wiąże gniazdo z adresem IP 127.0.0.1 i portem 12345 za pomocą server\_socket.bind(('127.0.0.1', 12345)).
5. Ustawia gniazdo w tryb nasłuchiwania na maksymalnie jedno połączenie przychodzące jednocześnie (server\_socket.listen(1)).
6. Wypisuje komunikat informujący, że serwer nasłuchuje na porcie 12345.
7. W nieskończonej pętli oczekuje na połączenia przychodzące:
   * Akceptuje połączenie od klienta za pomocą server\_socket.accept(), co zwraca nowe gniazdo klienta i adres klienta.
   * Wypisuje adres połączonego klienta.
   * Odbiera dane od klienta (do 1024 bajtów) za pomocą client\_socket.recv(1024).decode().
   * Jeśli dane zostały odebrane, wypisuje je i wydobywa pierwszą część danych (przy założeniu, że dane są rozdzielone średnikami).
   * Wysyła tę część danych z powrotem do klienta (client\_socket.send(packet\_no.encode())).
   * Zamyka połączenie z klientem (client\_socket.close()).
8. Funkcja start\_server() jest wywoływana na końcu, uruchamiając serwer.

Kod:

import socket  
  
def start\_server():  
 server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 server\_socket.bind(('127.0.0.1', 12345))  
 server\_socket.listen(1)  
 print("Server is listening on port 12345")  
  
 while True:  
 client\_socket, addr = server\_socket.accept()  
 print(f"Connection from {addr}")  
 data = client\_socket.recv(1024).decode()  
 if data:  
 print(f"Received data: {data}")  
 packet\_no = data.split(';')[0]  
 client\_socket.send(packet\_no.encode())  
 client\_socket.close()  
  
start\_server()

* 1. Encoder.py

Ten kod zawiera kilka funkcji do przetwarzania danych na różne sposoby, w tym podział na pakiety i dodawanie bitów parzystości oraz kodów CRC (Cyclic Redundancy Check). Oto szczegółowy opis każdej funkcji:

1. **Importowanie modułów:**
   * zlib: Wykorzystywany do obliczania wartości CRC32.
   * crcmod: Używany do generowania wartości CRC8 i CRC16.
2. **Funkcja to\_packets(data: str, packet\_size: int):**
   * Dzieli dane (data) na pakiety o określonym rozmiarze (packet\_size).
   * Zwraca listę pakietów.
3. **Funkcja to\_even\_bit(data):**
   * Liczy liczbę jedynek w danych (data).
   * Dodaje bit parzystości na końcu danych: 0 jeśli liczba jedynek jest parzysta, 1 jeśli jest nieparzysta.
   * Zwraca dane z dodanym bitem parzystości.
4. **Funkcja to\_crc8(data):**
   * Tworzy obiekt CRC8 z predefiniowanymi ustawieniami (crc-8).
   * Aktualizuje obiekt CRC8 danymi (data) przekonwertowanymi na bajty.
   * Oblicza wartość CRC8 i dodaje ją na końcu danych w postaci 8-bitowego ciągu binarnego.
   * Zwraca dane z dodaną wartością CRC8.
5. **Funkcja to\_crc16(data):**
   * Podobnie jak to\_crc8, ale dla CRC16 (crc-16).
   * Oblicza wartość CRC16 i dodaje ją na końcu danych w postaci 16-bitowego ciągu binarnego.
   * Zwraca dane z dodaną wartością CRC16.
6. **Funkcja to\_crc32(data):**
   * Oblicza wartość CRC32 z danych (data) zakodowanych jako ciąg bajtów.
   * Dodaje wartość CRC32 na końcu danych w postaci 32-bitowego ciągu binarnego.
   * Zwraca dane z dodaną wartością CRC32.

Te funkcje są używane do podziału danych na mniejsze części (pakiety), dodawania bitów parzystości oraz dołączania kodów kontrolnych CRC8, CRC16 i CRC32, które pomagają w wykrywaniu błędów podczas transmisji danych.

Kod:

import zlib  
import crcmod  
  
  
def to\_packets(data: str, packet\_size: int):  
 return [data[i:i+packet\_size] for i in range(0, len(data), packet\_size)]  
  
  
def to\_even\_bit(data):  
 count = data.count("1")  
 if count % 2 == 0:  
 parity\_bit = "0"  
 else:  
 parity\_bit = "1"  
 return data + parity\_bit  
  
  
def to\_crc8(data):  
 crc8 = crcmod.predefined.Crc('crc-8')  
 crc8.update(int(data, 2).to\_bytes((len(data) + 7) // 8, byteorder='big'))  
 crc\_value = crc8.crcValue  
 encoded\_sequence = data + format(crc\_value, '08b')  
 return encoded\_sequence  
  
  
def to\_crc16(data):  
 crc16 = crcmod.predefined.Crc('crc-16')  
 crc16.update(int(data, 2).to\_bytes((len(data) + 7) // 8, byteorder='big'))  
 crc\_value = crc16.crcValue  
 encoded\_sequence = data + format(crc\_value, '016b')  
 return encoded\_sequence  
  
  
def to\_crc32(data):  
 crc32 = zlib.crc32(data.encode())  
 return data + format(crc32 & 0xFFFFFFFF, '032b')

* 1. Klasa Decoder.py

Ten kod zawiera kilka funkcji do dekodowania danych, w tym łączenie pakietów oraz sprawdzanie i usuwanie bitów parzystości i kodów CRC (Cyclic Redundancy Check). Oto szczegółowy opis każdej funkcji:

1. **Importowanie modułów:**
   * crcmod: Używany do generowania wartości CRC8 i CRC16.
   * zlib: Wykorzystywany do obliczania wartości CRC32.
2. **Funkcja from\_packets(packets: list):**
   * Łączy listę pakietów w jeden ciąg znaków.
   * Zwraca połączone dane jako jeden ciąg znaków.
3. **Funkcja to\_even\_bit\_decode(binary\_sequence):**
   * Liczy liczbę jedynek w sekwencji binarnej (binary\_sequence).
   * Jeśli liczba jedynek jest parzysta, usuwa ostatni bit (bit parzystości) i zwraca pozostałą część sekwencji.
   * Jeśli liczba jedynek jest nieparzysta, zwraca None, co oznacza, że wystąpił błąd w transmisji.
4. **Funkcja to\_crc8\_decode(encoded\_sequence):**
   * Tworzy obiekt CRC8 z predefiniowanymi ustawieniami (crc-8).
   * Dzieli zakodowaną sekwencję (encoded\_sequence) na dane i otrzymaną wartość CRC.
   * Oblicza wartość CRC8 dla danych i porównuje ją z otrzymaną wartością CRC.
   * Jeśli wartości są zgodne, zwraca dane; w przeciwnym razie zwraca None.
5. **Funkcja to\_crc16\_decode(encoded\_sequence):**
   * Podobnie jak to\_crc8\_decode, ale dla CRC16 (crc-16).
   * Dzieli zakodowaną sekwencję na dane i otrzymaną wartość CRC.
   * Oblicza wartość CRC16 dla danych i porównuje ją z otrzymaną wartością CRC.
   * Jeśli wartości są zgodne, zwraca dane; w przeciwnym razie zwraca None.
6. **Funkcja to\_crc32\_decode(encoded\_sequence):**
   * Dzieli zakodowaną sekwencję na dane i otrzymaną wartość CRC.
   * Oblicza wartość CRC32 dla danych i porównuje ją z otrzymaną wartością CRC.
   * Jeśli wartości są zgodne, zwraca dane; w przeciwnym razie zwraca None.

Te funkcje są używane do dekodowania danych, sprawdzania ich integralności poprzez bity parzystości oraz kody CRC (CRC8, CRC16, CRC32), a także łączenia pakietów w jeden ciąg znaków.

import crcmod  
import zlib  
  
  
def from\_packets(packets: list):  
 return ''.join([el for packet in packets for el in str(packet)])  
  
  
def to\_even\_bit\_decode(binary\_sequence):  
 count\_ones = binary\_sequence.count('1')  
 if count\_ones % 2 == 0:  
 return binary\_sequence[:-1]  
 else:  
 return None  
  
  
def to\_crc8\_decode(encoded\_sequence):  
 crc8 = crcmod.predefined.Crc('crc-8')  
 data = encoded\_sequence[:-8]  
 received\_crc = int(encoded\_sequence[-8:], 2)  
 crc8.update(data.encode())  
 calculated\_crc = crc8.crcValue  
 if calculated\_crc == received\_crc:  
 return data  
 else:  
 return None  
  
  
def to\_crc16\_decode(encoded\_sequence):  
 crc16 = crcmod.predefined.Crc('crc-16')  
 data = encoded\_sequence[:-16]  
 received\_crc = int(encoded\_sequence[-16:], 2)  
 crc16.update(data.encode())  
 calculated\_crc = crc16.crcValue  
 if calculated\_crc == received\_crc:  
 return data  
 else:  
 return None  
  
  
def to\_crc32\_decode(encoded\_sequence):  
 encoded\_length = len(encoded\_sequence)  
 crc\_length = 32  
 data\_length = encoded\_length - crc\_length  
 data = encoded\_sequence[:data\_length]  
 received\_crc = int(encoded\_sequence[data\_length:], 2)  
 calculated\_crc = zlib.crc32(data.encode()) & 0xFFFFFFFF  
 if calculated\_crc == received\_crc:  
 return data  
 else:  
 return None

* 1. Klasa receiver.py

Ten kod tworzy serwer TCP, który odbiera zakodowane pakiety od klienta, dekoduje je przy użyciu funkcji z modułu Decoder, a następnie łączy te pakiety w całość. Oto szczegółowy opis, co robi każda część kodu:

1. **Importowanie modułów:**
   * socket: Umożliwia komunikację sieciową.
   * sleep z modułu time: Umożliwia wprowadzenie opóźnień w działaniu programu.
   * Decoder: Używany do dekodowania otrzymanych danych (zakłada się, że moduł Decoder zawiera funkcje dekodujące, jak to\_even\_bit\_decode i from\_packets).
2. **Tworzenie gniazda serwera:**
   * s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM): Tworzy gniazdo TCP.
   * s.bind(('127.0.0.1', 12345)): Wiąże gniazdo z lokalnym adresem IP 127.0.0.1 i portem 12345.
   * s.listen(5): Ustawia gniazdo w tryb nasłuchiwania na maksymalnie pięć połączeń oczekujących.
3. **Akceptowanie połączenia:**
   * c, addr = s.accept(): Akceptuje połączenie przychodzące i zwraca nowe gniazdo klienta (c) oraz adres klienta (addr).
4. **Odbieranie danych:**
   * Inicjalizuje pustą listę packets do przechowywania odebranych i zdekodowanych pakietów.
   * Inicjalizuje zmienną data jako '0' do rozpoczęcia pętli.
5. **Pętla odbierania danych:**
   * data = c.recv(1024).decode(): Odbiera dane od klienta w blokach po 1024 bajty i dekoduje je.
   * Jeśli dane są puste lub nie zawierają znaku ;, przechodzi do następnej iteracji.
   * packet\_no, packet = data.split(';')[0:2]: Dzieli dane na numer pakietu (packet\_no) i zawartość pakietu (packet).
   * Wypisuje odebrany numer pakietu i zawartość pakietu.
   * decoded = Decoder.to\_even\_bit\_decode(packet): Dekoduje zawartość pakietu za pomocą funkcji to\_even\_bit\_decode z modułu Decoder.
   * Jeśli numer pakietu (packet\_no) jest większy lub równy długości listy packets i decoded nie jest None:
     + Dodaje zdekodowany pakiet do listy packets.
     + Wprowadza opóźnienie 2 sekund (sleep(2)).
     + Wysyła numer pakietu z powrotem do klienta (c.send(packet\_no.encode())).
6. **Po zakończeniu pętli:**
   * print('Otrzymane dane', Decoder.from\_packets(packets)): Łączy wszystkie pakiety w jeden ciąg znaków za pomocą Decoder.from\_packets i wypisuje wynik.
   * c.close(): Zamyka połączenie z klientem.

W ten sposób serwer odbiera zakodowane pakiety, dekoduje je, gromadzi i w końcu wypisuje połączone dane.

Kod:

import socket  
from time import sleep  
import Decoder  
  
s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
s.bind(('127.0.0.1', 12345))  
s.listen(5)  
  
c, addr = s.accept()  
packets = []  
  
data = '0'  
while data != '':  
 data = c.recv(1024).decode()  
 if data == '' or ';' not in data:  
 continue  
 packet\_no, packet = data.split(';')[0:2]  
 print('Otrzymano', packet\_no, packet)  
 decoded = Decoder.to\_even\_bit\_decode(packet)  
 if int(packet\_no) >= len(packets) and decoded is not None:  
 packets.append(decoded)  
 sleep(2)  
 c.send(packet\_no.encode())  
  
print('Otrzymane dane', Decoder.from\_packets(packets))  
c.close()

* 1. Klasa sender.py

Ten kod implementuje symulację transmisji danych w sieci z wykorzystaniem różnych metod kodowania błędów oraz modeli kanału transmisji, takich jak Binary Symmetric Channel (BSC) i model Gilberta-Elliotta (GE). Transmisja jest testowana zarówno z ARQ (Automatic Repeat reQuest), jak i bez niego. Poniżej znajduje się szczegółowy opis działania poszczególnych części kodu:

1. **Importowanie modułów:**
   * socket: Do komunikacji sieciowej.
   * sleep: Do wprowadzenia opóźnień.
   * Encoder: Zakłada się, że zawiera funkcje do kodowania danych.
   * numpy (jako np): Do operacji matematycznych i losowych.
   * Enum: Do definiowania enumeracji.
   * random: Do generowania losowych sekwencji binarnych.
   * komm: Do symulacji kanałów transmisji.
2. **Definicje klas i funkcji:**
   * Model: Enumeracja zawierająca dwa modele kanałów: BSC i GE.
   * GilbertElliott: Klasa implementująca model Gilberta-Elliotta z określonymi prawdopodobieństwami przejścia między stanami dobrym (G) i złym (B) oraz prawdopodobieństwem błędu w stanie złym.
   * simulate\_errors(data: str, model: Model, ...): Funkcja symulująca błędy w danych przy użyciu wybranego modelu kanału (BSC lub GE).
   * send\_packet(packet, packet\_no, model, error\_probability, choice, retry\_no=1): Funkcja wysyłająca pakiet danych do serwera, kodująca dane za pomocą wybranej metody (bit parzystości, CRC8, CRC16, CRC32) i symulująca błędy. Jeśli odpowiedź serwera jest prawidłowa, zwraca True, w przeciwnym razie powtarza próbę wysłania pakietu.
   * generate\_random\_binary\_sequence(length): Generuje losową sekwencję binarną o określonej długości.
   * calculate\_ber(sent\_data, received\_data): Oblicza wskaźnik błędu bitowego (BER) między wysłanymi a odebranymi danymi.
   * calculate\_group\_errors(sent\_data, received\_data, group\_size): Oblicza liczbę błędów grupowych w danych.
   * transmission\_with\_arq(packets, model, error\_probability, choice): Funkcja realizująca transmisję z ARQ, wysyła pakiety i oczekuje na potwierdzenie odbioru od serwera, obsługuje retransmisję w przypadku błędów.
   * transmission\_without\_arq(packets, model, error\_probability, choice): Funkcja realizująca transmisję bez ARQ, wysyła pakiety bez oczekiwania na potwierdzenie odbioru.
3. **Ustawienia i inicjalizacja:**
   * max\_retries, max\_error\_frames, error\_probability, model, prob\_good\_to\_bad, prob\_bad\_to\_good, choice, sequence\_length, group\_size: Parametry konfiguracyjne symulacji.
   * s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM): Tworzy gniazdo TCP.
   * s.settimeout(0.1): Ustawia czas oczekiwania na odpowiedź.
   * s.connect(('127.0.0.1', 12345)): Łączy się z serwerem.
4. **Generowanie danych i pakietów:**
   * random\_binary\_sequence = generate\_random\_binary\_sequence(sequence\_length): Generuje losową sekwencję binarną.
   * example\_data = random\_binary\_sequence: Przykładowe dane do transmisji.
   * packets = Encoder.to\_packets(example\_data, 32): Dzieli dane na pakiety o rozmiarze 32 bitów.
5. **Transmisja z ARQ:**
   * received\_data\_arq, error\_count\_arq, total\_retries, independent\_errors\_arq = transmission\_with\_arq(packets, model, error\_probability, choice): Wykonuje transmisję z ARQ i zbiera statystyki błędów.
6. **Transmisja bez ARQ:**
   * received\_data\_no\_arq, independent\_errors\_no\_arq = transmission\_without\_arq(packets, model, error\_probability, choice): Wykonuje transmisję bez ARQ i zbiera statystyki błędów.
7. **Zamknięcie gniazda i wyniki:**
   * s.close(): Zamknięcie połączenia z serwerem.
   * Wypisanie wyników transmisji z ARQ i bez ARQ, w tym wskaźnika błędu bitowego (BER), liczby błędów grupowych i niezależnych błędów.

Kod ten jest kompleksową symulacją, która pozwala na ocenę różnych metod kodowania i modeli kanału transmisji w kontekście ich skuteczności w przekazywaniu danych oraz obsługi błędów.

Kod:

import socket  
from time import sleep  
import Encoder  
import numpy as np  
from enum import Enum  
import random  
import komm  
  
  
class Model(Enum):  
 BSC = 'BSC'  
 GE = 'GE'  
  
  
class GilbertElliott:  
 def \_\_init\_\_(self, p\_good\_to\_bad, p\_bad\_to\_good, p\_error\_bad):  
 self.p\_good\_to\_bad = p\_good\_to\_bad  
 self.p\_bad\_to\_good = p\_bad\_to\_good  
 self.p\_error\_bad = p\_error\_bad  
 self.state = 'G'  
  
 def transmit(self, bit):  
 if self.state == 'G':  
 if np.random.rand() < self.p\_good\_to\_bad:  
 self.state = 'B'  
 else:  
 if np.random.rand() < self.p\_bad\_to\_good:  
 self.state = 'G'  
  
 if self.state == 'B' and np.random.rand() < self.p\_error\_bad:  
 return '1' if bit == '0' else '0'  
 return bit  
  
  
def simulate\_errors(data: str, model: Model, error\_probability=0.001, prob\_good\_to\_bad=0.1, prob\_bad\_to\_good=0.1):  
 if model == Model.BSC:  
 np.random.seed(1)  
 bsc = komm.BinarySymmetricChannel(error\_probability)  
 y = bsc([int(x) for x in list(data)])  
 return ''.join(map(str, y))  
 elif model == Model.GE:  
 np.random.seed(1)  
 ge = GilbertElliott(prob\_good\_to\_bad, prob\_bad\_to\_good, error\_probability)  
 y = [ge.transmit(bit) for bit in data]  
 return ''.join(y)  
  
  
def send\_packet(packet, packet\_no, model: Model, error\_probability, choice, retry\_no=1):  
 if retry\_no > max\_retries:  
 return False  
 try:  
 if choice == 0:  
 encoded = Encoder.to\_even\_bit(packet)  
 elif choice == 1:  
 encoded = Encoder.to\_crc8(packet)  
 elif choice == 2:  
 encoded = Encoder.to\_crc16(packet)  
 elif choice == 3:  
 encoded = Encoder.to\_crc32(packet)  
 data = f'{packet\_no};{simulate\_errors(encoded, model, error\_probability)}'  
 s.send(data.encode())  
 sleep(s.gettimeout())  
 response = s.recv(1024).decode()  
 if ';' in response:  
 response\_packet\_no, received\_packet = response.split(';', 1)  
 if response\_packet\_no == str(packet\_no):  
 return True  
 return send\_packet(packet, packet\_no, model, error\_probability, choice, retry\_no + 1)  
 except socket.error:  
 return send\_packet(packet, packet\_no, model, error\_probability, choice, retry\_no + 1)  
  
  
def generate\_random\_binary\_sequence(length):  
 return ''.join(random.choice(['0', '1']) for \_ in range(length))  
  
  
def calculate\_ber(sent\_data, received\_data):  
 errors = sum(1 for sent, received in zip(sent\_data, received\_data) if sent != received)  
 total\_bits = len(sent\_data)  
 if total\_bits == 0:  
 return 0  
 return errors / total\_bits  
  
  
def calculate\_group\_errors(sent\_data, received\_data, group\_size):  
 group\_errors = 0  
 for i in range(0, len(sent\_data), group\_size):  
 if sent\_data[i:i + group\_size] != received\_data[i:i + group\_size]:  
 group\_errors += 1  
 return group\_errors  
  
def transmission\_with\_arq(packets, model, error\_probability, choice):  
 received\_data = []  
 error\_count = 0  
 total\_retries = 0  
  
 for idx, p in enumerate(packets):  
 retries = 0  
 while not send\_packet(p, packet\_no=idx, model=model, error\_probability=error\_probability, choice=choice,  
 retry\_no=retries):  
 retries += 1  
 total\_retries += 1  
 if retries > max\_retries:  
 error\_count += 1  
 break  
 if error\_count > max\_error\_frames:  
 print("Przekroczono maksymalną liczbę błędnych ramek.")  
 break  
 try:  
 response = s.recv(1024).decode()  
 if ';' in response:  
 packet\_no, received\_packet = response.split(';', 1)  
 received\_data.append(received\_packet)  
 except socket.timeout:  
 error\_count += 1  
 if error\_count > max\_error\_frames:  
 print("Przekroczono maksymalną liczbę błędnych ramek.")  
 break  
  
 # Obliczenie BER oraz błędów grupowych  
 received\_data\_flat = ''.join(received\_data[:len(example\_data)])  
 independent\_error\_count = sum(1 for sent, received in zip(example\_data, received\_data\_flat) if sent != received)  
 return received\_data\_flat, error\_count, total\_retries, independent\_error\_count  
  
def transmission\_without\_arq(packets, model, error\_probability, choice):  
 received\_data = []  
  
 for idx, p in enumerate(packets):  
 if choice == 0:  
 encoded = Encoder.to\_even\_bit(p)  
 elif choice == 1:  
 encoded = Encoder.to\_crc8(p)  
 elif choice == 2:  
 encoded = Encoder.to\_crc16(p)  
 elif choice == 3:  
 encoded = Encoder.to\_crc32(p)  
 data = simulate\_errors(encoded, model, error\_probability)  
 received\_data.append(data)  
  
 # Obliczenie BER oraz błędów grupowych  
 received\_data\_flat = ''.join(received\_data[:len(example\_data)])  
 independent\_error\_count = sum(1 for sent, received in zip(example\_data, received\_data\_flat) if sent != received)  
 return received\_data\_flat, independent\_error\_count  
  
  
# Ustawienia  
max\_retries = 0  
max\_error\_frames = 20 #parametr maksymalnej liczby błędnych ramek  
error\_probability = 0.0001 # Przykładowe prawdopodobieństwo błędu  
model = Model.GE # Wybór modelu kanału  
prob\_good\_to\_bad = 0.1 # Prawdopodobieństwo przejścia ze stanu dobrego do złego (dla GE)  
prob\_bad\_to\_good = 0.1 # Prawdopodobieństwo przejścia ze stanu złego do dobrego (dla GE)  
choice = 1 # 0-Bit parzystosci, 1-crc8, 2-crc16, 3-crc32  
sequence\_length = 2048  
group\_size = 4  
  
s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
s.settimeout(0.1)  
try:  
 s.connect(('127.0.0.1', 12345))  
except socket.timeout:  
 print("Timeout: Could not connect to the server.")  
 s.close()  
 exit()  
  
random\_binary\_sequence = generate\_random\_binary\_sequence(sequence\_length)  
example\_data = random\_binary\_sequence  
packets = Encoder.to\_packets(example\_data, 32)  
  
# Transmisja z ARQ  
received\_data\_arq, error\_count\_arq, total\_retries, independent\_errors\_arq = transmission\_with\_arq(packets, model, error\_probability, choice)  
ber\_arq = calculate\_ber(example\_data, received\_data\_arq)  
group\_errors\_arq = calculate\_group\_errors(example\_data, received\_data\_arq, group\_size)  
  
# Transmisja bez ARQ  
received\_data\_no\_arq, independent\_errors\_no\_arq = transmission\_without\_arq(packets, model, error\_probability, choice)  
ber\_no\_arq = calculate\_ber(example\_data, received\_data\_no\_arq)  
group\_errors\_no\_arq = calculate\_group\_errors(example\_data, received\_data\_no\_arq, group\_size)  
  
# Zamknij gniazdo  
s.close()  
  
# Wyniki  
print("Transmisja z ARQ:")  
print(f"Bit Error Rate (BER): {ber\_arq}")  
print(f"Liczba błędów grupowych: {group\_errors\_arq}")  
print(f"Liczba błędów niezależnych: {independent\_errors\_arq}")  
  
print("\nTransmisja bez ARQ:")  
print(f"Bit Error Rate (BER): {ber\_no\_arq}")  
print(f"Liczba błędów grupowych: {group\_errors\_no\_arq}")  
print(f"Liczba błędów niezależnych: {independent\_errors\_no\_arq}")

# Opis programu

## Użyte biblioteki zewnętrzne

* **Komm** [8]: Biblioteka Komm została użyta w celu symulacji transmisji przy użyciu modelu BSC
* **Crcmod** [9]: Biblioteka została użyta w celu usprawnienia pracy z kodami CRC
* **Zlib** [10]**:** Biblioteka została użyta w celu usprawnienia pracy z kodem CRC32

# Bibliografia

|  |  |
| --- | --- |
| [1] | „ScienceDirect,” [Online]. Available: https://www.sciencedirect.com/topics/computer-science/automatic-repeat-request. |
| [2] | „GeeksForGeeks,” [Online]. Available: https://www.geeksforgeeks.org/what-is-arq-automatic-repeat-request/. |
| [3] | „GeeksForGeeks,” [Online]. Available: https://www.geeksforgeeks.org/stop-and-wait-arq/. |
| [4] | „ScienceDirect,” [Online]. Available: https://www.sciencedirect.com/topics/mathematics/binary-symmetric-channel. |
| [5] | „Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Binary\_symmetric\_channel. |
| [6] | „Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Burst\_error. |
| [7] | „Wikipedia,” [Online]. Available: https://pl.wikipedia.org/wiki/Kontrola\_parzysto%C5%9Bci. |
| [8] | „Komm,” [Online]. Available: https://komm.dev/. |
| [9] | „Crcmod Sourceforge,” [Online]. Available: https://crcmod.sourceforge.net/. |
| [10] | „Zlib,” [Online]. Available: https://docs.python.org/3/library/zlib.html. |