

PSIR

Projekt

Symulacja rozproszonej gry Penney's Ante

Jakub Strzelczyk **325325**

Jakub Sierocki **325318**

Jakub Grzechnik **325278**

20 stycznia 2025

Spis treści

1.	Wprowadzenie	2
1.1.	Cel projektu	2
1.2.	Założenia projektowe	2
2.	Realizacja projektu	3
2.1.	Specyfikacja ALP	3
2.2.	Opis implementacji serwera	5
2.3.	Opis implementacji klienta	8
3.	Podsumowanie	9



**Wydział Elektroniki
i Technik Informacyjnych**

POLITECHNIKA WARSZAWSKA

1. Wprowadzenie

1.1. Cel projektu

Celem projektu jest opracowanie i implementacja rozproszonej aplikacji symulującej grę Penney's Ante. Gra polega na wybraniu przez uczestników wzorców rzutów monet - ciągów składających się z H (heads) oraz T (tails), a zwycięzcą jest gracz, którego wzorzec pojawi się jako pierwszy w sekwencji rzutów monet.



Rysunek 1. Obraz poglądowy - gra Penney's Ante

1.2. Założenia projektowe

Poniżej przedstawiamy założenia dotyczące naszego projektu, opisujące funkcjonalność poszczególnych aspektów oraz użyte narzędzia:

Serwer:

- Zarządzanie grami w sposób rozproszony.
- Rozsyłanie wyników rzutów monet do klientów.
- Zbieranie raportów od klientów i obliczanie statystyk.
- Implementacja w języku C na systemie Linux.
- Niedozwolone jest korzystanie z zewnętrznych bibliotek, platform czy middleware.

Klient (węzeł IoT):

- Wprowadzanie wybranego wzorca.
- Odbieranie wyników rzutów monet od serwera i identyfikacja momentu dopasowania wzorca.
- Raportowanie wyników do serwera.
- Symulacja w środowisku Arduino.
- Wykorzystanie standardowego API Arduino oraz rozszerzeń PSIR.

Protokół ALP:

- Własnoręcznie zdefiniowany protokół binarny.
- Platformo- i językowo niezależny.
- Opiera się na protokole UDP, co zapewnia prostotę i wydajność w systemach IoT.
- Definicja formatu wiadomości na poziomie bitowym.
- Minimalizacja zużycia pamięci i ilości przesyłanych danych.

2. Realizacja projektu

2.1. Specyfikacja ALP

Protokół ALP został zbudowany w oparciu o protokół UDP, który nie gwarantuje niezawodnego ani uporządkowanego dostarczania pakietów. W celu zapewnienia możliwie pewnego dostarczenia danych, każda wiadomość ALP zawiera numer sekwencyjny **seq_num**, który jest reprezentowany na 8 bitach. Nadawca, po wysłaniu wiadomości, oczekuje na pakiet ACK (potwierdzenie) z takim samym numerem sekwencyjnym przez z góry ustalony czas. Jeśli w tym czasie nie nadaje się ACK, wysłana wiadomość jest retransmitowana. W przypadku dalszego braku potwierdzenia, procedura retransmisji jest powtarzana aż do osiągnięcia maksymalnego limitu prób lub otrzymania prawidłowego ACK. Mechanizm ten pozwala na minimalne zapewnienie spójności komunikacji, choć nie zastępuje bardziej rozbudowanych rozwiązań warstwy transportowej - jak TCP.

Pojedyncza wiadomość ALP może składać się minimalnie z 14 bitów, a maksymalnie z 21 bitów, w zależności od zawartości (tzw. *payload*). Ogólna struktura przedstawia się następująco:

wiadomość = kod, długość payloadu, numer sekwencyjny, payload

Tabela 1. Typy wiadomości w protokole ALP

Typ wiadomości	Kod (3 bity)	Długość payloadu (3 bity)	Payload (0 - 7 bitów)
REGISTER (rejestracja klienta na serwerze)	000	000	–
ACK (potwierdzenie)	001	000	–
WANT_PLAY	010	001	1 – chce grać, 0 – nie chce grać
START_GAME (wiadomość o starcie gry)	011	000	–
PATTERN_SENT (wiadomość zawierająca wzorzec klienta)	100	001–111	np. 10001 = HTTTH
TOSS_MSG (wiadomość o wylosowaniu)	101	001	1 – H, 0 – T
END (zakończenie rundy z informacją o wyniku)	110	001	1 – wygrana, 0 – porażka
EXIT (kończenie gry)	111	000	–

W powyższej tabeli:

- **Kod (3 bity)** definiuje rodzaj wiadomości.
- **Długość payloadu (3 bity)** określa, ile bajtów danych jest załączone w części Payload.
- **Payload (0 - 7 bitów)** to właściwa treść wiadomości, której obecność i interpretacja zależy od długości payloadu oraz typu wiadomości.

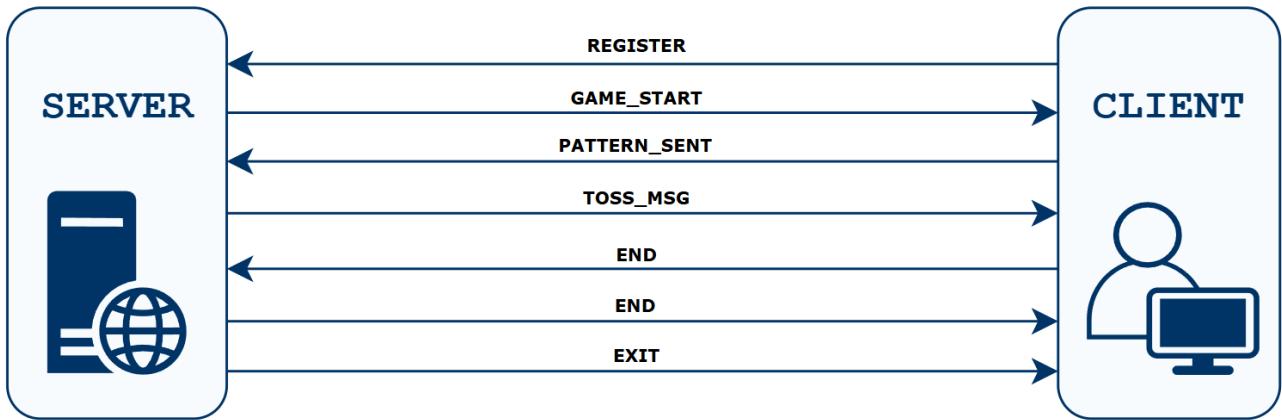
Podczas implementacji natrafiliśmy na pewne trudności, dlatego ostatecznie zdecydowaliśmy się zrezygnować z mechanizmu retransmisji. Zamiast tego zastosowaliśmy prostsze metody, takie jak wykrywanie błędów

oraz efektywniejsze zarządzanie sesjami, co okazało się wystarczające dla niezawodności naszej gry. W trakcie implementacji zrezygnowaliśmy z użycia wiadomości WANT_PLAY i ACK.

W tak uproszczonej formie każda wiadomość ALP wygląda następująco:

- **Bajt 0** (8 bitów):
 - Bity (7..5): typ wiadomości (TYPE, wartości 0–7),
 - Bity (4..2): długość danych (LEN, wartości 0–7),
 - Bity (1..0): niewykorzystywane (padding).
- **Bajt 1** (8 bitów): Numer sekwencyjny (`seq_num`), używany do potwierdzeń (ACK) i prostego śledzenia komunikacji.
- **Bajt 2** (opcjonalnie): Pojawia się tylko wtedy, gdy LEN jest większe od 0. Może zawierać do 8 bitów danych, ale w praktyce wykorzystuje się tylko LEN bitów.

Poniżej przedstawiamy diagram (flow wiadomości) w naszym programie:



Rysunek 2. Diagram wiadomości protokołu ALP

- **Rejestracja gracza:** Gracz wysyła wiadomość REGISTER.
- **Rozpoczęcie gry:** Serwer wysyła wiadomość START_GAME.
- **Wysyłanie wzorców:** Gracz wysyła wiadomość PATTERN_SENT.
- **Rzut:** Serwer wysyła wiadomość TOSS_MSG.
- **Zakończenie gry:**
 - Gracz wysyła wiadomość END z 1, która oznacza znalezienie wzorca.
 - Serwer wysyła każdemu graczowi wiadomość END z wynikiem.
- **Zamknięcie sesji:** Serwer wysyła wiadomość EXIT.

Implementacja protokołu ALP w języku C:

Listing 1. struktura ALP w języku C

```

1 #define REGISTER 0
2 #define ACK 1
3 #define WANT_PLAY 2
4 #define START_GAME 3
5 #define PATTERN_SENT 4
6 #define TOSS_MSG 5
7 #define END 6
8 #define EXIT 7
9
10 typedef struct
11 {
12     uint8_t type;
13     uint8_t len;
14     uint8_t seq_num;
15     uint8_t data[MAX_DATA_LEN];
16 } ALPMessage;
  
```

Każde pole tej struktury odpowiada specyfikacji opisanej w tabeli na początku rozdziału 2.1. Typ i długość wiadomości zajmują odpowiednie bity w polu **type**.

W celu przedstawienia i przesłania wiadomości w formacie bitowym, a także ich interpretacji z formatu binarnego na strukturę stworzyliśmy dwie funkcje, których opis przedstawiamy poniżej.

Funkcja kodująca wiadomość ALP

Funkcja `encodeALPMessage` odpowiada za kodowanie wiadomości w odpowiednim formacie binarnym:

Listing 2. kod źródłowy funkcji `encodeALPMessage`

```
1 void encodeALPMessage(ALPMessage *msg, uint8_t *buffer)
2 {
3     buffer[0] = (msg->type << 5) | (msg->len << 2); // Typ i dlugosc
4     buffer[1] = msg->seq_num; // Numer sekwencyjny
5
6     if (msg->len > 0)
7     {
8         uint8_t data = 0;
9         for (int i = 0; i < msg->len; i++)
10        {
11            data |= (msg->data[i] << (msg->len - i - 1)); // Przesunięcie danych
12        }
13        buffer[2] = data; // Dane w trzecim bajcie
14    }
15}
```

Funkcja ta zapewnia odpowiednią strukturę danych przed ich wysłaniem poprzez protokół UDP.

Funkcja dekodująca wiadomość ALP

Funkcja `decodeALPMessage` dekoduje odebrane dane z formatu binarnego:

Listing 3. kod źródłowy funkcji `decodeALPMessage`

```
1 void decodeALPMessage(uint8_t *buffer, ALPMessage *msg)
2 {
3     msg->type = (buffer[0] >> 5) & 0x07; // Typ wiadomosci
4     msg->len = (buffer[0] >> 2) & 0x07; // Dlugosc danych
5     msg->seq_num = buffer[1]; // Numer sekwencyjny
6
7     if (msg->len > 0)
8     {
9         uint8_t data = buffer[2];
10        for (int i = 0; i < msg->len; i++)
11        {
12            msg->data[i] = (data >> (msg->len - i - 1)) & 0x01; // Odczyt danych
13        }
14    }
15}
```

Dzięki tej funkcji klient może interpretować różne typy wiadomości, takie jak START_GAME, TOSS_MSG czy END.

2.2. Opis implementacji serwera

Serwer obsługujący rozproszoną grę Penney's Ante został napisany w języku C i opiera się na protokole UDP do komunikacji z klientami. Jego główne zadanie to zarządzanie rejestracją klientów, zbieranie ich wzorców oraz koordynacja rzutów monetą i analiza wyników. Implementacja obejmuje kilka kluczowych elementów, które zostały opisane poniżej.

Inicjalizacja serwera

Serwer korzysta z gniazda UDP, które jest tworzone za pomocą funkcji `socket(AF_INET, SOCK_DGRAM, 0)`. Po utworzeniu gniazda jest ono powiązane z adresem IP i portem (zdefiniowanym jako PORT) za pomocą funkcji `bind`. W przypadku błędu podczas tworzenia lub powiązania gniazda aplikacja kończy działanie, wyświetlając

odpowiedni komunikat.

Rejestracja klientów

Serwer przez 20 sekund po uruchomieniu oczekuje na wiadomości typu REGISTER od klientów. Każdy klient, który się zarejestruje, jest dodawany do tablicy klientów (`clients`). Dodanie nowego klienta wymaga sprawdzenia, czy jego adres (IP i port) nie jest już w tablicy. Informacje o zarejestrowanych klientach, takie jak adres IP i port, są wypisywane na konsoli. Jeśli żaden klient się nie zarejestruje, serwer kończy działanie.

```
UDP server started on port 8888
Waiting for REGISTER message for 20 seconds...
REGISTER message received.

Client Table:
+---+---+---+
| # | IP Address | Port |
+---+---+---+
| 1 | 192.168.56.104 | 8889 |
+---+---+---+

REGISTER message received.

Client Table:
+---+---+---+
| # | IP Address | Port |
+---+---+---+
| 1 | 192.168.56.104 | 8889 |
| 2 | 192.168.56.105 | 8889 |
+---+---+---+
```

Rysunek 3. Inicjalizacja serwera i rejestracja klientów

Obsługa gier

Serwer realizuje sekwencję gier, których liczba jest określona przez stałą `GAME_ROUNDS`. Rozpoczęcie gry jest sygnalizowane wiadomością `START_GAME`, którą serwer wysyła do wszystkich zarejestrowanych klientów. Po tym etapie serwer zbiera wzorce przesyłane przez klientów i przechowuje je w ich strukturach. Następnie rozpoczyna się symulacja rzutów monetą generowanych za pomocą funkcji `rand()`, a wyniki rzutów są przesyłane do klientów w wiadomościach `TOSS_MSG`. Serwer przechowuje historię ostatnich pięciu rzutów. Za znalezienie wzorca odpowiada klient. W momencie, gdy otrzymuje od serwera wiadomość `TOSS_MSG`, porównuje ją ze swoim wzorcem. Jeśli wzorzec się zgadza, klient wysyła do serwera komunikat `END`. Następnie serwer, otrzymawszy tę wiadomość, weryfikuje, czy dany klient rzeczywiście wygrał, kończy grę i przesyła klientom wyniki rundy.

```

2 clients registered. Starting the game.

Starting game round 1
Pattern received from client 1: TTHTT
Pattern received from client 2: HHHTT
-----| # | IP Address | Pattern |
-----| 1 | 192.168.56.104 | TTHTT |
-----| 2 | 192.168.56.105 | HHHTT |

Toss 1: T
Toss 2: H
Toss 3: T
Toss 4: T
Toss 5: T | Last 5 tosses: TTHTT
Toss 6: T | Last 5 tosses: HTTTT
Toss 7: T | Last 5 tosses: TTTTT
Toss 8: H | Last 5 tosses: TTTHH
Toss 9: H | Last 5 tosses: TTHHH
Toss 10: H | Last 5 tosses: THHHH
Toss 11: T | Last 5 tosses: THHHT
Toss 12: T | Last 5 tosses: HHHTT

Winner:
-----| Client 2 |
-----| IP | 192.168.56.105 |
-----| Pattern | HHHTT |
-----| Tosses | 12 |

```

Rysunek 4. Start gry, odczyt wzorców i pierwsza runda gry

```

Starting game round 10
-----| # | IP Address | Pattern |
-----| 1 | 192.168.56.104 | TTHTT |
-----| 2 | 192.168.56.105 | HHHTT |

Toss 1: H
Toss 2: H
Toss 3: T
Toss 4: T
Toss 5: H | Last 5 tosses: HHTTH
Toss 6: T | Last 5 tosses: HTTHT
Toss 7: T | Last 5 tosses: TTHTT

Winner:
-----| Client 1 |
-----| IP | 192.168.56.104 |
-----| Pattern | TTHTT |
-----| Tosses | 7 |

```

Rysunek 5. Dziesiąta runda gry

Statystyki

Po zakończeniu wszystkich gier serwer oblicza i wyświetla statystyki, w tym:

- liczbę graczy
- długość wzorców
- liczbę rozegranych rund
- średnią liczbę rzutów potrzebnych do wyłonienia zwycięzcy
- liczbę wysłanych i odebranych wiadomości
- prawdopodobieństwo wygranej dla każdego wzorca

Game Statistics:	
Parameter	Value
Number of players	2
Length of the patterns	5
Game rounds played	20
Average number of tosses	17.15
Messages sent	768
Messages received	26

Winning Probabilities:	
Pattern	Winning Probability [%]
TTHTT	50.00
HHHTT	50.00

Rysunek 6. Statystyki gry

Statystyki są przedstawione w formie tabel, co ułatwia ich analizę.

Kodowanie i dekodowanie wiadomości

Serwer wykorzystuje protokół ALP (Application Layer Protocol) do komunikacji z klientami. Wiadomości ALP są binarnie kodowane i dekodowane za pomocą funkcji `encodeALPMessage` oraz `decodeALPMessage`. Struktura wiadomości ALP jest dokładnie opisana w rozdziale 2.1.

2.3. Opis implementacji klienta

Klient gry Penney's Ante został zaimplementowany w języku C, uruchamiany jest przez emulator Arduino i komunikuje się z serwerem za pomocą protokołu UDP. Jego głównymi zadaniami są rejestracja w systemie, przekazywanie swojego wzorca, odbieranie wyników rzutów monetą oraz raportowanie wyników.

Inicjalizacja klienta

Klient inicjalizuje połączenie sieciowe przy użyciu biblioteki `ZsutEthernetUDP`. Adres serwera (`serverIP`) oraz port (`serverPort`) są zdefiniowane w kodzie jako stałe. Ponadto klient konfiguruje swoje piny wejściowe (`ZsutPinMode`), które służą do odczytywania wzorca z fizycznych przełączników. Następnie klient tworzy wiadomość `REGISTER`, która informuje serwer o jego gotowości do uczestnictwa w grze. Wiadomość ta jest kodowana przy pomocy protokołu ALP i przesyłana do serwera.

Obsługa wzorca klienta

Wzorzec klienta jest odczytywany z pinów GPIO (zdefiniowanych w pliku `infile.txt`) za pomocą funkcji `readPatternFromPins`. Wzorzec składa się z pięciu bitów, które reprezentują symbole H (head) i T (tails). Każdy bit wzorca jest przetwarzany na postać binarną i przechowywany w tablicy `pattern_bin`. Gotowy wzorzec jest przechowywany w postaci tekstowej (`pattern`) oraz binarnej, co pozwala na łatwą manipulację i transmisję.

Obsługa wiadomości serwera

Klient działa w pętli głównej (`loop`), w której odbiera wiadomości od serwera i odpowiednio na nie reaguje. Obsługiwane typy wiadomości to:

- `START_GAME`: Klient odczytuje swój wzorzec i przesyła go do serwera w wiadomości `PATTERN_SENT`.
- `TOSS_MSG`: Klient odbiera wynik rzutu monetą (H lub T) i aktualizuje historię ostatnich pięciu rzutów. Jeśli sekwencja rzutów pasuje do wzorca, klient przesyła do serwera wiadomość `END`, informując o swojej wygranej.
- `END`: Klient otrzymuje wynik gry (wygrana lub przegrana) i wyświetla odpowiedni komunikat na konsoli.
- `EXIT`: Klient otrzymuje sygnał zakończenia gry i kończy działanie.

Pliki środowiskowe

Plik `infile.txt` definiuje powiązania pomiędzy pinami GPIO a ich funkcjami w grze. Zawiera on mapowanie przełączników i decyzji do odpowiednich sygnałów. Dla przykładu, piny D1–D5 odpowiadają bitom wzorca gry, gdzie wartość 1 oznacza H (head), a wartość 0 oznacza T (tails). Przykładowy fragment pliku:

Listing 4. plik `infile.txt`

```
1 + qPattern1 ,action ,D1  # Pattern bit 1
2 + qPattern2 ,action ,D2  # Pattern bit 2
3 + qPattern3 ,action ,D3  # Pattern bit 3
4 + qPattern4 ,action ,D4  # Pattern bit 4
5 + qPattern5 ,action ,D5  # Pattern bit 5
6 : 10, qPattern1, 1 # H
7 : 20, qPattern2, 1 # H
8 : 30, qPattern2, 0 # T
9 : 40, qPattern2, 0 # T
10: 50, qPattern2, 0 # T
```

Wartości te są odczytywane przez funkcję `readPatternFromPins`, która interpretuje stany fizycznych przełączników, a następnie konwertuje je na wzorzec gry. Taka konfiguracja pozwala na łatwe dostosowanie zachowania klienta bez modyfikacji kodu źródłowego.

Kodowanie i dekodowanie wiadomości

Klient korzysta z protokołu ALP (Application Layer Protocol) do wymiany wiadomości z serwerem. Wiadomości są kodowane i dekodowane za pomocą funkcji `encodeALPMessage` i `decodeALPMessage`. Struktura wiadomości ALP dokładnie opisana jest w rozdziale 2.1.

Demonstracja

```
| # | IP Address | Port |
| 1 | 192.168.56.104 | 8889 |
REGISTER message received.

Client Table:
| # | IP Address | Port |
| 1 | 192.168.56.104 | 8889 |
| 2 | 192.168.56.105 | 8889 |

2 clients registered. Starting the game.

Starting game round 1
Pattern received from client 1: TTHTT
Pattern received from client 2: HHHTT

| # | IP Address | Pattern |
| 1 | 192.168.56.104 | TTHTT |
| 2 | 192.168.56.105 | HHHTT |

Toss 1: H
Toss 2: H
Toss 3: H
Toss 4: H
Toss 5: H | Last 5 tosses: HHHHH
Toss 6: H | Last 5 tosses: HHHHH
Toss 7: H | Last 5 tosses: HHHHH
Toss 8: T | Last 5 tosses: HHHTH
Toss 9: H | Last 5 tosses: HHHTH
Toss 10: H | Last 5 tosses: HHHTH
Toss 11: T | Last 5 tosses: HTHTH
Toss 12: T | Last 5 tosses: THHTT
Toss 13: H | Last 5 tosses: HHTTH
Toss 14: T | Last 5 tosses: HTHTH
Toss 15: T | Last 5 tosses: THHTT
Pattern matched! Sending result to server.

| You won! |
```

```
GPIO
[Z0:0x03ff Z1:0x03ff Z2:0x03ff Z3:0x03ff Z4:0x03ffz
5:0x03ff
[D0:1 D1:0 D2:0 D3:1 D4:0 D5:0 D6:1 D7:1 D8:1 D
13:1

UART
UDP client started.
REGISTER message sent to server.
Game starting!
Toss: 1: H
Toss: 2: H
Toss: 3: H
Toss: 4: H
Toss: 5: H | Last 5 tosses: HHHHH
Toss: 6: H | Last 5 tosses: HHHHH
Toss: 7: H | Last 5 tosses: HHHHH
Toss: 8: T | Last 5 tosses: HHHTH
Toss: 9: H | Last 5 tosses: HHHTH
Toss: 10: H | Last 5 tosses: HHHTH
Toss: 11: T | Last 5 tosses: HTHTH
Toss: 12: T | Last 5 tosses: THHTT
Toss: 13: H | Last 5 tosses: HHTTH
Toss: 14: T | Last 5 tosses: HTHTH
Toss: 15: T | Last 5 tosses: THHTT
Pattern matched! Sending result to server.

| You lost! |
```

Rysunek 7. Pierwsza runda rozgrywki dla dwóch klientów

Rysunek 8. Ostatnia runda rozgrywki dla dwóch klientów

Po lewej stronie znajduje się serwer, a po prawej klienci.

3. Podsumowanie

Projekt pozwolił nam zdobyć i pogłębić wiedzę w zakresie projektowania systemów rozproszonych oraz efektywnej komunikacji w środowisku IoT. Nauczyliśmy się, jak tworzyć i implementować własny protokół komunikacyjny, dostosowany do specyficznych wymagań, takich jak minimalizacja zużycia zasobów i ograniczenie objętości przesyłanych danych. Zyskaliśmy praktyczne doświadczenie w programowaniu niskopoziomowym w

języku C oraz w pracy z protokołem UDP, co wymagało od nas znajomości mechanizmów zarządzania pakietami, takich jak numeracja sekwencyjna czy retransmisje.

Praca nad klientem w środowisku Arduino nauczyła nas, jak integrować urządzenia IoT z systemami komunikacyjnymi oraz jak efektywnie korzystać z dostępnych bibliotek i API w ograniczonych środowiskach sprzętowych. Ponadto, zdobyliśmy umiejętności analizy i wizualizacji statystyk rozgrywek, co pomogło nam lepiej zrozumieć działanie implementowanego systemu.

Projekt wymagał również od nas efektywnej współpracy zespołowej, zarządzania zadaniami oraz rozwiązywania problemów technicznych na różnych etapach realizacji. Dzięki temu nauczyliśmy się, jak podchodzić do złożonych problemów w sposób systematyczny i iteracyjny, co z pewnością będzie przydatne w przyszłych projektach.



**Wydział Elektroniki
i Technik Informacyjnych**

POLITECHNIKA WARSZAWSKA