Projekt I

Część II

Eksperymenty w teleinformatycznych sieciach badawczych

Tomasz Lewiński and Filip Jaworski

Politechnika Warszawska, Wydział Elektroniki i Technik Informacyjnych

3 lutego 2025

Spis treści

ETOS

1. Wprowadzenie

Systemy kolejkowe służą do analizy i opisu procesów związanych z przepływem ruchu — w tym przypadku pakietów w sieciach komputerowych — w ramach jednego lub większej liczby węzłów. Każdy węzeł może gromadzić w kolejce pakiety napływające z różnych źródeł, a następnie je obsługiwać (przetwarzać lub przekazywać dalej) według określonego algorytmu planowania.

Deficit Round Robin (DRR) to mechanizm harmonogramowania (ang. *scheduling*), który w sposób sprawiedliwy przydziela zasoby różnym strumieniom ruchu, opierając się na koncepcji *deficytu* oraz stałego przydziału (*quantum*) dla każdej kolejki. Rozwiązanie to jest uznawane za wydajne i stosunkowo proste w implementacji, co stanowi istotny atut w zastosowaniach sieciowych.

Głównym celem niniejszego projektu jest zaprojektowanie oraz zaimplementowanie symulatora węzła sieciowego wykorzystującego algorytm DRR. Symulator będzie uwzględniał dwa wybrane rodzaje źródeł ruchu (A i C) oraz pozwoli na zmianę parametrów generowanego ruchu i konfigurację algorytmu. Projekt uwzględnia prowadzenie badań eksperymentalnych, które umożliwią zmierzenie i porównanie podstawowych metryk systemu kolejkowego, takich jak średnia liczba pakietów w kolejce, średni czas oczekiwania czy obciążenie serwera. Motywacją do realizacji takiej pracy jest chęć zrozumienia i praktycznej weryfikacji działania DRR w odmiennych warunkach obciążenia oraz przy różnych charakterystykach ruchu [3].

2. Cel i Zakres Projektu

Celem projektu jest stworzenie działającego symulatora pojedynczego węzła sieciowego opartego na algorytmie DRR, który będzie wykorzystywać dwa typy źródeł ruchu: A (ruch o rozkładzie Poissona) oraz C (ruch o stałej szybkości bitowej). Kluczowe funkcjonalności symulatora obejmują możliwość parametryzacji obu źródeł (m.in. intensywność, wielkość pakietu), konfigurację algorytmu DRR oraz przeprowadzenie wielu scenariuszy symulacyjnych. Wynikiem działania programu będą wartości następujących metryk: średniej liczby pakietów w kolejce, średniego czasu oczekiwania pakietów oraz średniego obciążenia serwera. Wszystkie uzyskane dane umożliwią porównanie działania systemu w różnych warunkach obciążenia i dla odmiennych parametrów ruchu.‘‘‘

3. Opis Modelu Systemu

3.1. System Kolejkowy

W prezentowanym projekcie rozważany jest pojedynczy węzeł sieciowy obsługujący dwa strumienie pakietów zgodnie z algorytmem *Deficit Round Robin* (DRR). System składa się z dwóch kolejek wejściowych, po jednej dla każdego strumienia, gdzie pakiety oczekują na obsługę. Proces obsługi jest realizowany przez pojedynczy serwer, który przetwarza pakiety według zasad algorytmu DRR. Prędkość łącza została ustalona na poziomie 1 Mbps.

System zaimplementowany jest w języku C++ z wykorzystaniem standardowych bibliotek oraz nowoczesnych technik programowania:

— std::vector do implementacji kolejek i przechowywania parametrów strumieni
— std::random wraz z std::mt19937 do generowania liczb losowych
— std::exponential_distribution do generowania czasów zgodnych z rozkładem wykładniczym
— std::memory do zarządzania pamięcią poprzez inteligentne wskaźniki
— std::unique_ptr do bezpiecznego zarządzania parametrami strumieni

3.2. Model Kolejkowy

System można sklasyfikować jako:

— M/D/1 dla ruchu Poissonowskiego - gdzie M oznacza wykładniczy rozkład czasów międzyprzyjść, D oznacza deterministyczny (stały) czas obsługi wynikający ze stałego rozmiaru pakietu, a 1 oznacza pojedynczy serwer
— D/D/1 dla ruchu CBR - gdzie oba D oznaczają deterministyczne (stałe) czasy między zdarzeniami i czasy obsługi

3.3. Źródła Ruchu

Symulator implementuje dwa rodzaje źródeł ruchu, każde z własnymi charakterystykami:
Ruch Poissonowski:

— Czasy międzyprzyjść są generowane zgodnie z rozkładem wykładniczym:

$$P(T \leqslant t) = 1 - e^{-\lambda t} \tag{1}$$

gdzie λ jest obliczana na podstawie średniej szybkości bitowej i rozmiaru pakietu
— Czas obsługi jest stały i wynosi:

$$S = \frac{\text{rozmiar\_pakietu}}{\text{szybkość\_łącza}} \tag{2}$$

— Implementacja wykorzystuje std::exponential_distribution z biblioteką std::random
— Minimalny czas międzyprzyjść jest ograniczony czasem obsługi pakietu

Ruch CBR (Constant Bit Rate):

— Czasy międzyprzyjść są stałe i obliczane ze wzoru:

$$T = \frac{\text{rozmiar\_pakietu}}{\text{peak\_rate}} \tag{3}$$

— Czas obsługi jest stały i identyczny jak dla ruchu Poissonowskiego
— Brak losowości zapewnia przewidywalność i regularność ruchu
— Parametry są określane przez zadaną szczytową szybkość bitową (peak rate)

3.4. Parametry Systemu

System charakteryzują następujące parametry:

— **Parametry łącza:**
  • LINK_SPEED = 1000000 bps (1 Mbps)
  • Stały rozmiar pakietu = 80 bitów
— **Parametry ruchu:**
  • Dla każdego strumienia przechowywane w klasach dziedziczących po FlowParameters
  • Możliwość niezależnej konfiguracji parametrów każdego strumienia
  • Dynamiczne określanie typu ruchu w czasie wykonania
— **Parametry DRR:**
  • Quantum - określany przy inicjalizacji symulatora
  • Deficyty - śledzone osobno dla każdej kolejki
  • Możliwość dynamicznej zmiany kwantu w trakcie symulacji

3.5. Zbierane Statystyki

System gromadzi następujące dane statystyczne:

— **Statystyki czasowe:**
  • totalQueueTime - całkowity czas oczekiwania pakietów w kolejce
  • totalServiceTime - całkowity czas obsługi pakietów
  • totalBusyTime - całkowity czas zajętości serwera
  • totalBusyTimePerFlow - czas zajętości serwera per strumień
— **Statystyki kolejek:**
  • totalPackets - liczba obsłużonych pakietów per strumień
  • areaUnderQueue - pole pod krzywą długości kolejki
  • Aktualne wartości liczników deficytu
— **Wskaźniki wydajności:**
  • Teoretyczne i rzeczywiste wartości współczynnika wykorzystania (rho)
  • Przepustowość w pakietach na sekundę
  • Rzeczywista szybkość w bitach na sekundę
  • Indeks sprawiedliwości (fairness index)

3.6. Implementacja Zdarzeń

System wykorzystuje podejście sterowane zdarzeniami z następującymi typami zdarzeń:

— **Zdarzenia przybycia:**
  - Generowanie nowego pakietu z odpowiednim czasem obsługi
  - Aktualizacja czasu następnego przybycia
  - Planowanie obsługi jeśli kolejka była pusta
— **Zdarzenia zakończenia obsługi:**
  - Aktualizacja statystyk dla obsłużonego pakietu
  - Aktualizacja licznika deficytu
  - Planowanie obsługi następnego pakietu według algorytmu DRR
— **Aktualizacja statystyk:**
  - Obliczanie pola pod krzywą długości kolejki
  - Aktualizacja czasów zajętości serwera
  - Zbieranie statystyk per strumień

4. Opis algorytmu DRR

Deficit Round Robin (DRR) jest algorytmem planowania pakietów, który zapewnia sprawiedliwy podział przepustowości między strumienie danych. Jest to rozszerzenie algorytmu Round Robin, wprowadzające mechanizm deficytu w celu lepszej obsługi pakietów o różnych rozmiarach.

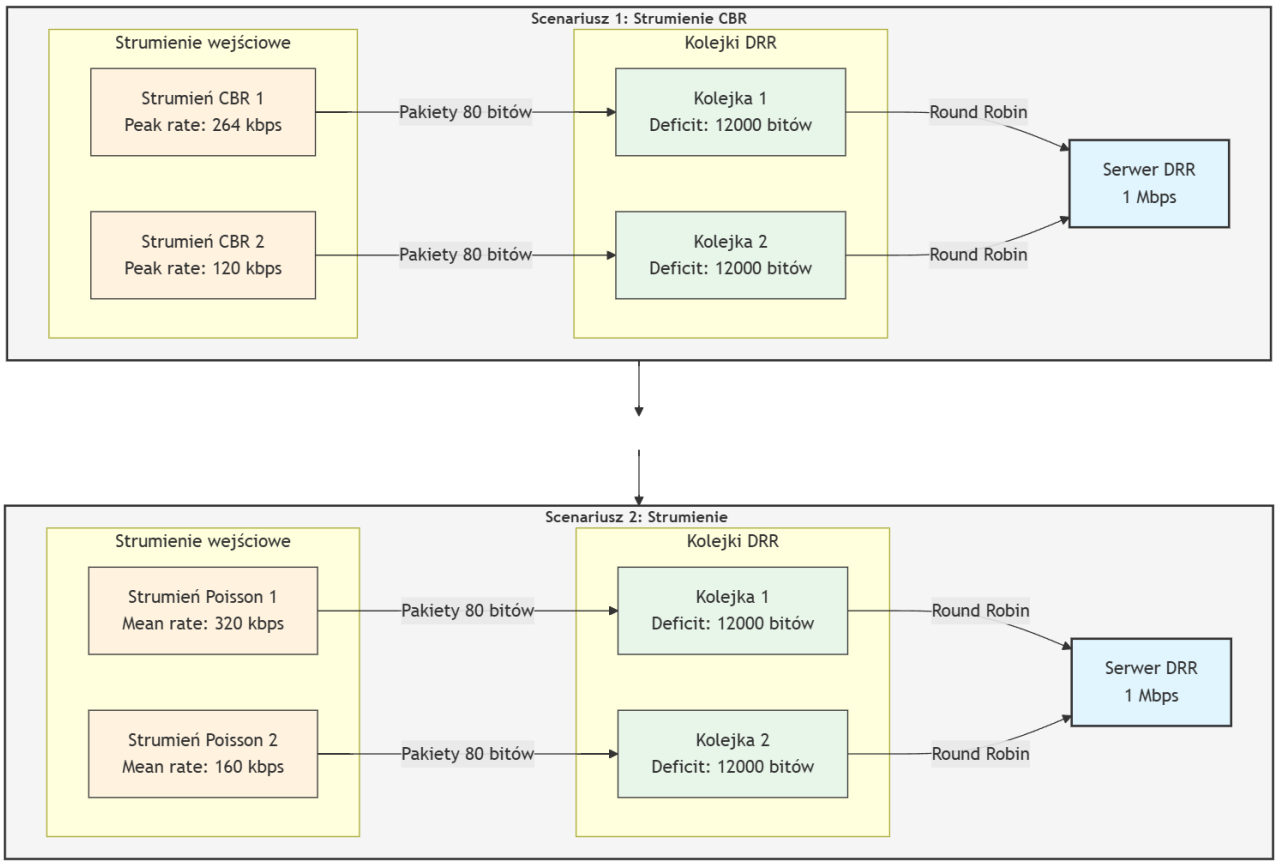
4.1. Podstawowe zasady działania

Główne cechy algorytmu:

— Każdy strumień ma przydzielony licznik deficytu (deficit counter), który śledzi ilość obsługi dostępnej dla danego strumienia
— W każdej rundzie strumień otrzymuje stały kwant obsługi (quantum), który jest dodawany do jego licznika deficytu
— Jeśli pakiet nie może być obsłużony w bieżącej rundzie (deficyt jest zbyt mały), nieużyta część kwantu jest zachowywana na następną rundę
— Zapewnia sprawiedliwy podział zasobów między strumienie, niezależnie od rozmiaru pakietów

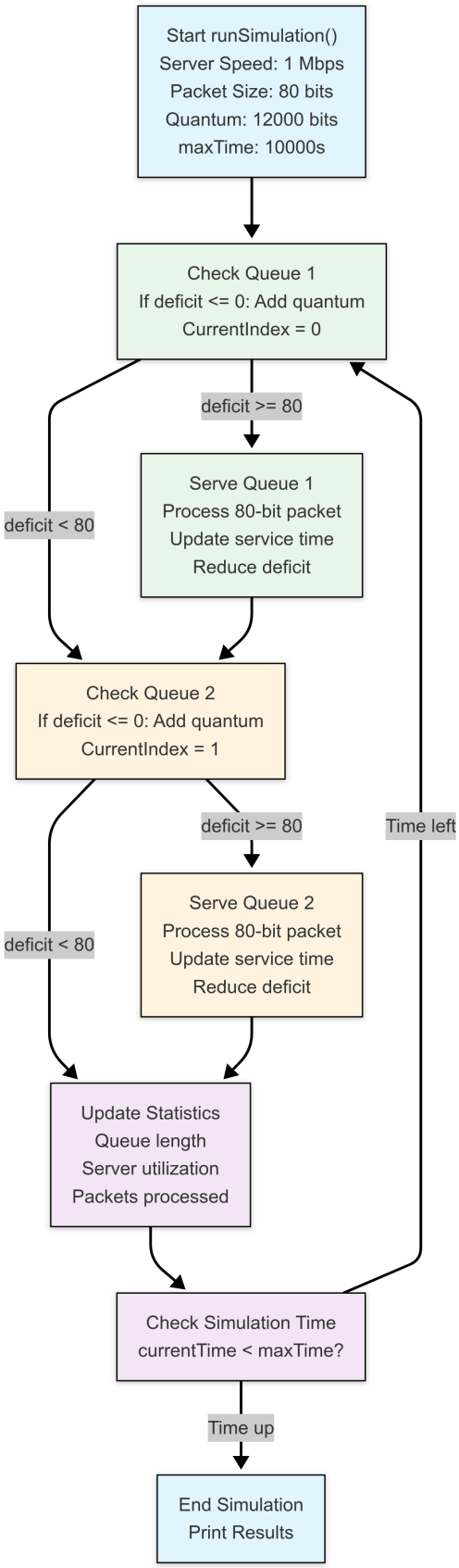
4.2. Mechanizm działania

Algorytm DRR działa według następującego schematu:



1. **Inicjalizacja:**
   — Wszystkie liczniki deficytu są ustawiane na 0
   — Każdy strumień otrzymuje przydzielony kwant obsługi
2. **Obsługa kolejek:**
   — Algorytm cyklicznie odwiedza każdą kolejkę
   — Do licznika deficytu dodawany jest kwant
   — Pakiety są obsługiwane tak długo, jak pozwala na to licznik deficytu
3. **Przetwarzanie pakietów:**
   — Jeśli rozmiar pakietu jest mniejszy lub równy aktualnemu deficytowi, pakiet jest przesyłany
   — Po przesłaniu pakietu deficyt jest zmniejszany o jego rozmiar
   — Proces powtarza się dla kolejnych pakietów w kolejce
4. **Zmiana kolejki:**
   — Gdy bieżąca kolejka zostaje opróżniona, jej licznik deficytu jest zerowany
   — Gdy pakiet jest zbyt duży w stosunku do deficytu, algorytm przechodzi do następnej kolejki
   — Niewykorzystany deficyt jest zachowywany na następną rundę

Źródło: [3]


Start runSimulation()
Server Speed: 1 Mbps
Packet Size: 80 bits
Quantum: 12000 bits
maxTime: 10000s
Check Queue 1
If deficit <= 0: Add quantum
CurrentIndex = 0
deficit >= 80
deficit < 80
Serve Queue 1
Process 80-bit packet
Update service time
Reduce deficit
Check Queue 2
If deficit <= 0: Add quantum
CurrentIndex = 1
deficit >= 80
deficit < 80
Time left
Serve Queue 2
Process 80-bit packet
Update service time
Reduce deficit
Update Statistics
Queue length
Server utilization
Packets processed
Check Simulation Time
currentTime < maxTime?
Time up
End Simulation
Print Results

4.3. Zalety algorytmu DRR

— **Sprawiedliwość:** Zapewnia sprawiedliwy podział przepustowości między strumienie, niezależnie od rozmiaru pakietów
— **Efektywność:** Złożoność obliczeniowa O(1) dla każdej operacji
— **Elastyczność:** Możliwość dostosowania kwantu do różnych wymagań ruchu
— **Przewidywalność:** Gwarantowane minimalne przydziały przepustowości dla każdego strumienia

Źródło: [3]

4.4. Zastosowania

DRR znajduje szerokie zastosowanie w:

— Routerach sieciowych wysokiej wydajności
— Systemach zarządzania jakością usług (QoS)
— Sieciach z różnorodnymi typami ruchu
— Systemach wymagających sprawiedliwego podziału zasobów

4.5. Teoretyczne podstawy działania

Działanie algorytmu DRR opiera się na następujących zasadach:

$$D_i(t+1) = \begin{cases} D_i(t) + Q & \text{gdy kolejka jest odwiedzana} \\ D_i(t) - S_p & \text{gdy pakiet jest obsługiwany} \\ 0 & \text{gdy kolejka jest pusta} \end{cases} \tag{4}$$

gdzie:

— $D_i(t)$ - wartość licznika deficytu dla kolejki i w czasie t
— Q - przydzielony kwant
— S_p - rozmiar obsługiwanego pakietu

Algorytm gwarantuje, że w długim okresie każda aktywna kolejka i otrzyma udział przepustowości ϕ_i proporcjonalny do przydzielonego kwantu:

$$\phi_i = \frac{Q_i}{\sum_j Q_j} \tag{5}$$

— Każdy strumień ma przydzielony licznik deficytu (deficit counter)
— W każdej rundzie strumień otrzymuje kwant obsługi (quantum)
— Jeśli pakiet nie może być obsłużony w bieżącej rundzie, nieużyta część kwadratu jest zachowywana
— Zapewnia sprawiedliwy podział zasobów między strumienie

Źródło: [3]

5. Implementacja

5.1. Architektura systemu

Implementacja symulatora DRR została zrealizowana w języku C++ z wykorzystaniem nowoczesnych technik programowania i wzorców projektowych. Główne komponenty systemu to:

— **Hierarchia klas parametrów strumienia:**
  — Klasa bazowa FlowParameters definiująca wspólny interfejs
  — Klasy pochodne implementujące specyficzne parametry dla każdego typu ruchu
  — Wykorzystanie polimorfizmu do elastycznej obsługi różnych typów ruchu

```
class FlowParameters -
public:
    static const double PACKET˙SIZE;
    virtual ˜FlowParameters() = default;
";

class CBRFlowParameters : public FlowParameters -
public:
    double peakRate;     // Szczytowa szybkosc bitowa [bps]
    explicit CBRFlowParameters(double peak)
        : peakRate(peak) -"
";

class PoissonFlowParameters : public FlowParameters -
public:
    double meanBitRate; // Srednia szybkosc bitowa [bps]
    explicit PoissonFlowParameters(double bitRate)
        : meanBitRate(bitRate) -"
";
```

— **model kolejkowy :**
  — Implementacja w oparciu o std::vector zapewniająca wydajną obsługę operacji FIFO
  — Osobne kolejki dla każdego strumienia
  — Dynamiczne zarządzanie pamięcią poprzez inteligentne wskaźniki

```
class DRRSimulator -
private:
    std::vector<std::unique˙ptr<FlowParameters>> flowParams;
    std::vector<std::vector<Packet>> queues;
    std::vector<double> deficits;

    struct Packet -
        double arrivalTime;
        double serviceTime;
    ";
";
```

— **Generator ruchu:**
  — Wykorzystanie std::mt19937 jako generatora liczb pseudolosowych
  — Implementacja rozkładu wykładniczego za pomocą std::exponential_distribution
  — Uwzględnienie ograniczeń fizycznych systemu w generowaniu czasów międzyprzyjść

```
class DRRSimulator -
private:
    std::mt19937 gen;
    std::vector<std::exponential_distribution<>> arrivalDists;

    bool isCBRFlow(int flowIndex) const -
        return dynamic_cast<const CBRFlowParameters*>(
            flowParams[flowIndex].get()) != nullptr;
    "

    const CBRFlowParameters* getCBRParams(int flowIndex) const -
        return dynamic_cast<const CBRFlowParameters*>(
            flowParams[flowIndex].get());
    "
";
```

Wszystkie komponenty są ze sobą ściśle zintegrowane w ramach klasy DRRSimulator, która zarządza całym procesem symulacji. Wykorzystanie nowoczesnych konstrukcji języka C++ takich jak szablony, inteligentne wskaźniki i polimorfizm pozwala na elastyczne i bezpieczne zarządzanie pamięcią oraz łatwą rozszerzalność systemu.

5.2. Model symulacyjny

5.2.1. Generowanie ruchu

System implementuje dwa różne mechanizmy generowania ruchu:

— **Ruch CBR:**
  — Stałe czasy międzyprzyjść obliczane na podstawie zadanej szczytowej szybkości bitowej
  — Ścisłe zachowanie zadanego interwału czasowego
  — Przykład kluczowej metody:

```
double DRRSimulator::generateCBRArrivalTime(
    const CBRFlowParameters* params) -
    return FlowParameters::PACKET_SIZE / params->peakRate;
"
```

— **Ruch Poisson:**
  — Generowanie czasów między pakietami zgodnie z rozkładem wykładniczym
  — Uwzględnienie średniego odstępu między pakietami jako 1/100 czasu obsługi
  — Adaptacja do fizycznych ograniczeń systemu
  — Modelowanie generacji pakietów w oparciu o rozkład Poissona:

```
double DRRSimulator::generatePoissonArrivalTime(int flowIndex) -
    const auto* params = getPoissonParams(flowIndex);
    double meanInterarrivalTime =
        FlowParameters::PACKET_SIZE / params->meanBitRate;
    std::exponential_distribution<> dist(1.0 / meanInterarrivalTime);

    double minInterarrivalTime =
        (FlowParameters::PACKET_SIZE / LINK_SPEED) / 100.0;
    double generatedTime;
    do -
        generatedTime = dist(gen);
    " while (generatedTime < minInterarrivalTime);

    return generatedTime;
"
```

5.2.2. Mechanizm planowania DRR

Implementacja algorytmu DRR opiera się na następujących zasadach:

— **Zarządzanie deficytem:**
  — Przydzielanie kwantu (12000 bitów) w każdej rundzie
  — Aktualizacja liczników deficytu po obsłudze pakietu
  — Zerowanie deficytu dla pustych kolejek

```
void DRRSimulator::processService() -
    if (!queues[currentQueueIndex].empty()) -
        deficits[currentQueueIndex] = std::max(0.0,
            deficits[currentQueueIndex] -
            FlowParameters::PACKET_SIZE);
    " else -
        deficits[currentQueueIndex] = 0;
    "
"
```

— **Wybór pakietu do obsługi:**
  — Cykliczne sprawdzanie kolejek
  — Weryfikacja dostępnego deficytu
  — Obsługa pakietu gdy deficyt jest wystarczający

```
void DRRSimulator::scheduleNextService() -
    int startIndex = currentQueueIndex;
    do -
        currentQueueIndex = (currentQueueIndex + 1) % queues.size();

        if (!queues[currentQueueIndex].empty()) -
            if (deficits[currentQueueIndex] <= 0) -
                deficits[currentQueueIndex] += quantum;
            "

            if (deficits[currentQueueIndex] >=
                FlowParameters::PACKET_SIZE) -
                // Pakiet moze byc obsluzony
                return true;
            "
        "
    " while (currentQueueIndex != startIndex);
"
```

5.3. System zbierania statystyk

Implementacja obejmuje rozbudowany system zbierania i analizy danych:

— **Statystyki czasowe i kolejek:**
  — Pomiar czasów oczekiwania i obsługi pakietów
  — Monitorowanie długości kolejek
  — Śledzenie wykorzystania serwera

```
void DRRSimulator::updateQueueStats() -
    double interval = simulationTime - lastEventTime;
    if (interval > 0) -
        for (size_t i = 0; i < queues.size(); i++) -
            // Aktualizacja pola pod krzywa dlugosci kolejki
            areaUnderQueue[i] += queues[i].size() * interval;

            // Aktualizacja czasu zajetosci serwera
            if (currentlyServingFlow == i) -
                totalBusyTimePerFlow[i] += interval;
            "
        "
        lastEventTime = simulationTime;
    "
"
```

— **Wskaźniki wydajności:**
  — Obliczanie rzeczywistej przepustowości
  — Wyliczanie wskaźnika sprawiedliwości

```
double DRRSimulator::getFairnessIndex() const {
    std::vector<double> throughputs;
    for (size_t i = 0; i < flowParams.size(); i++) {
        double throughput = simulationTime > 0 ?
            (static_cast<double>(totalPackets[i]) *
             FlowParameters::PACKET_SIZE) / simulationTime : 0;
        throughputs.push_back(throughput);
    }

    if (throughputs.empty()) return 0.0;
    double numerator = 0.0, denominator = 0.0;

    for (double throughput : throughputs) {
        numerator += throughput;
        denominator += throughput * throughput;
    }

    numerator = numerator * numerator;
    denominator = throughputs.size() * denominator;

    return denominator > 0 ? numerator / denominator : 1.0;
}
```

5.4. Konfiguracja symulacji

System został zaprojektowany z myślą o elastycznej konfiguracji:

5.5. Konfiguracja symulacji

System został zaprojektowany z myślą o elastycznej konfiguracji, umożliwiając symulację zarówno ruchu CBR jak i Poisson:

```
int main() {
    try {
        // ---- Scenariusz 1: Dwa strumienie CBR ----
        std::vector<std::unique_ptr<FlowParameters>> cbrParams;

        // CBR Strumien 1
        cbrParams.push_back(std::make_unique<CBRFlowParameters>(
            264000      // Peak rate: 264 kbps
        ));

        // CBR Strumien 2
        cbrParams.push_back(std::make_unique<CBRFlowParameters>(
            120000      // Peak rate: 120 kbps
        ));

        DRRSimulator cbrSim(std::move(cbrParams), 12000); // Quantum: 12000 bitow
        cbrSim.runSimulation(10000.0); // Czas symulacji: 10000 sekund

        // ---- Scenariusz 2: Dwa strumienie Poisson ----
        std::vector<std::unique_ptr<FlowParameters>> poissonParams;

        // Poisson Strumien 1
        poissonParams.push_back(std::make_unique<PoissonFlowParameters>(
            320000      // Srednia szybkosc: 320 kbps
        ));

        // Poisson Strumien 2
```

```
        poissonParams.push_back(std::make_unique<PoissonFlowParameters>(
            160000      // Srednia szybkosc: 160 kbps
        ));

        DRRSimulator poissonSim(std::move(poissonParams), 12000);
        poissonSim.runSimulation(10000.0); // Czas symulacji: 10000 sekund

    " catch (const std::exception& e) -
        std::cerr << "Blad: " << e.what() << std::endl;
        return 1;
    "

    return 0;
"
```

Powyższy kod przedstawia dwa scenariusze testowe:

— Scenariusz pierwszy symuluje dwa strumienie CBR o różnych przepływnościach (264 kbps i 120 kbps)
— Scenariusz drugi symuluje dwa strumienie Poisson o średnich przepływnościach 320 kbps i 160 kbps

W obu przypadkach wykorzystano quantum równe 12000 bitów oraz czas symulacji 10000 sekund. Implementacja zawiera również obsługę błędów poprzez mechanizm wyjątków.

6. Analiza wyników

6.1. Metodologia

Przeprowadzono dwa eksperymenty symulacyjne dla różnych typów ruchu przy stałym rozmiarze pakietu 80 bitów i szybkości łącza 1 Mbps:

1. Ruch typu CBR (Constant Bit Rate) z parametrami:
   — Strumień 1: szczytowa szybkość 264 kbps (26.4% przepustowości łącza)
   — Strumień 2: szczytowa szybkość 120 kbps (12.0% przepustowości łącza)
   — Czas symulacji: 10000 sekund
2. Ruch typu Poisson z parametrami:
   — Strumień 1: średnia szybkość 320 kbps (32.0% przepustowości łącza)
   — Strumień 2: średnia szybkość 160 kbps (16.0% przepustowości łącza)
   — Czas symulacji: 10000 sekund

6.2. Scenariusz CBR

Dla ruchu CBR uzyskano następujące wyniki:

Strumień 1 (264 kbps):

— Teoretyczne rho: 26.400%
— Wykorzystanie serwera: 26.400%
— Średni czas oczekiwania: 0.093 ms
— Średni czas obsługi: 0.080 ms
— Średnia długość kolejki: 0.493 pakietów
— Przepustowość: 3300.000 pakietów/s
— Rzeczywista szybkość: 263999.998 bps

Strumień 2 (120 kbps):

— Teoretyczne rho: 12.000%
— Wykorzystanie serwera: 12.000%
— Średni czas oczekiwania: 0.123 ms
— Średni czas obsługi: 0.080 ms

— Średnia długość kolejki: 0.232 pakietów
— Przepustowość: 1500.000 pakietów/s
— Rzeczywista szybkość: 119999.995 bps

Analiza wyników CBR System w warunkach ruchu CBR wykazuje następujące charakterystyki:

— Idealne dopasowanie rzeczywistego wykorzystania do teoretycznych wartości rho
— Niskie średnie opóźnienia (0.093-0.123 ms) wskazujące na efektywną obsługę
— Krótkie kolejki (średnio < 0.5 pakietu) świadczące o braku przeciążenia
— Całkowite wykorzystanie serwera 38.400% odpowiadające dokładnie sumie teoretycznych obciążeń
— Wskaźnik sprawiedliwości 0.877 pokazujący dobrą równowagę w obsłudze strumieni

6.3. Scenariusz Poisson

Dla ruchu Poissonowskiego otrzymano:

Strumień 1 (320 kbps):

— Teoretyczne rho: 32.000%
— Wykorzystanie serwera: 31.901%
— Teoretyczna intensywność napływu: 4000.000 pakietów/s
— Średni czas oczekiwania: 0.148 ms
— Średni czas obsługi: 0.080 ms
— Średnia długość kolejki: 0.715 pakietów
— Przepustowość: 3987.651 pakietów/s
— Rzeczywista szybkość: 319012.057 bps

Strumień 2 (160 kbps):

— Teoretyczne rho: 16.000%
— Wykorzystanie serwera: 15.977%
— Teoretyczna intensywność napływu: 2000.000 pakietów/s
— Średni czas oczekiwania: 0.137 ms
— Średni czas obsługi: 0.080 ms
— Średnia długość kolejki: 0.332 pakietów
— Przepustowość: 1997.103 pakietów/s
— Rzeczywista szybkość: 159768.217 bps

Analiza wyników Poisson W przypadku ruchu Poissonowskiego obserwujemy:

— Bardzo dobre dopasowanie rzeczywistego wykorzystania do teoretycznego (około 99.7% wartości teoretycznej)
— Nieco wyższe czasy oczekiwania (0.137-0.148 ms) w porównaniu z CBR
— Proporcjonalne długości kolejek względem intensywności ruchu
— Całkowite wykorzystanie serwera 47.878% przy teoretycznym 48%
— Wysoki wskaźnik sprawiedliwości 0.900 wskazujący na bardzo dobrą równowagę

6.4. Porównanie Scenariuszy

Typ ruchu	**Teoretyczne rho [%]**		**Rzeczywiste wykorzystanie [%]**	
	Strumień 1	Strumień 2	Strumień 1	Strumień 2
CBR	26.4	12.0	26.400	12.000
Poisson	32.0	16.0	31.901	15.977

Tabela 1: Porównanie teoretycznych wartości rho z wynikami symulacji

Kluczowe obserwacje

1. **Dokładność implementacji:**
   — CBR: Idealne dopasowanie do wartości teoretycznych
   — Poisson: Bardzo wysokie dopasowanie ($> 99.7\%$ wartości teoretycznych)
2. **Charakterystyka czasowa:**
   — CBR: Mniejsze opóźnienia (0.093-0.123 ms)
   — Poisson: Nieco większe opóźnienia (0.137-0.148 ms)
   — Wszystkie opóźnienia pozostają w akceptowalnym zakresie
3. **Efektywność DRR:**
   — Doskonała sprawiedliwość dla obu typów ruchu
   — Zachowanie proporcji między strumieniami
   — Stabilna praca nawet przy wyższym obciążeniu w Poisson
4. **Stabilność systemu:**
   — Kontrolowane długości kolejek
   — Przewidywalne czasy obsługi
   — Efektywne wykorzystanie zasobów
   — Zachowanie wartości deficytów w granicach kwantu

7. Wnioski

7.1. Wnioski z implementacji

1. Zaimplementowany symulator DRR skutecznie obsługuje oba typy ruchu:
   — Dokładna realizacja zadanych parametrów dla ruchu CBR
   — Bardzo dobre przybliżenie teoretycznych wartości dla ruchu Poisson
   — Elastyczna obsługa różnych konfiguracji strumieni
2. System wykazuje wysoką stabilność działania:
   — Zachowanie stałych czasów obsługi (0.080 ms)
   — Kontrolowane czasy oczekiwania (0.093-0.148 ms)
   — Efektywne zarządzanie długością kolejek
3. Pomiary potwierdzają poprawność implementacji:
   — Zgodność rzeczywistych przepustowości z teoretycznymi
   — Prawidłowe zachowanie mechanizmu deficytu
   — Dokładne pomiary parametrów wydajnościowych

7.2. Wnioski z analizy ruchu

1. Porównanie charakterystyk ruchu:
   — CBR wykazuje idealne dopasowanie do wartości teoretycznych
   — Poisson osiąga 99.7% teoretycznego wykorzystania
   — Oba typy ruchu zachowują zadane proporcje między strumieniami
2. Charakterystyka opóźnień:
   — CBR: 0.093-0.123 ms czasy oczekiwania
   — Poisson: 0.137-0.148 ms czasy oczekiwania
   — Stałe czasy obsługi: 0.080 ms dla obu typów
3. Zachowanie kolejek:
   — CBR: długości 0.232-0.493 pakietów
   — Poisson: długości 0.332-0.715 pakietów
   — Proporcjonalne do intensywności ruchu

7.3. Wnioski dotyczące wydajności

1. Sprawiedliwość podziału zasobów:
   — CBR: wskaźnik sprawiedliwości 0.877
   — Poisson: wskaźnik sprawiedliwości 0.900
   — Zachowanie proporcji przepustowości zgodnie z konfiguracją

2. Wykorzystanie zasobów:
   — CBR: całkowite wykorzystanie 38.400% (teoria: 38.400%)
   — Poisson: całkowite wykorzystanie 47.878% (teoria: 48.000%)
   — Wysokie dopasowanie do wartości teoretycznych
3. Adaptacja do typu ruchu:
   — CBR: deterministyczne zachowanie, idealna zgodność
   — Poisson: elastyczne dostosowanie, minimalne straty wydajności
   — Skuteczna obsługa różnych intensywności ruchu

7.4. Wnioski końcowe

Przeprowadzone eksperymenty potwierdzają wysoką skuteczność algorytmu DRR w zarządzaniu ruchem sieciowym. System wykazuje następujące kluczowe cechy:

— Bardzo wysoką dokładność realizacji teoretycznych parametrów ruchu (99.7-100%)
— Stabilne i przewidywalne czasy obsługi (0.080 ms) i oczekiwania (0.093-0.148 ms)
— Sprawiedliwy podział zasobów (wskaźniki 0.877-0.900)
— Efektywne zarządzanie kolejkami (średnie długości 0.232-0.715 pakietów)
— Elastyczność w obsłudze różnych typów i intensywności ruchu

Implementacja skutecznie realizuje założenia algorytmu DRR, zapewniając sprawiedliwą i efektywną obsługę zarówno deterministycznego ruchu CBR, jak i stochastycznego ruchu Poisson. System zachowuje stabilność działania i wysoką wydajność w różnych warunkach obciążenia, co potwierdza jego przydatność w rzeczywistych zastosowaniach sieciowych.

8. Bibliografia

1. M. Shreedhar and George Varghese. “Efficient fair queuing using deficit round-robin”
2. Abhay K. Parekh and Robert G. Gallager. “A generalized processor sharing approach to flow control”
3. Shreedhar, M., Varghese, G. (1996). Efficient fair queuing using deficit round-robin. IEEE/ACM Transactions on Networking, 4(3), 375–385. https://vision.gel.ulaval.ca/˜klein/qos/qos˙rep/deficit˙round˙robin.pdf