Lekcja 4. Wektory bram i wektory modułów

W lekcji 4 wyjaśnione zostają zagadnienia korzystania z wektorów modułów oraz wektorów bram ułatwiających budowę dużych symulacji.

Bardzo często zdarza się, że w symulacji znajdują się moduły spełniające te same funkcje i podłączone do tych samych modułów. Wypadałoby wtedy deklarować dla każdego z nich osobny moduł i inicjować każdy. W przypadku małych symulacji nie byłby to problem, ale gdy należy podłączyć np. 20 klientów do sieci to staje się to kłopotliwe. W takich sytuacjach najlepiej jest wykorzystać wektor modułu. Zawierał on będzie jakąś liczbę (określaną przez nas) pojedynczych podmodułów.
Ponadto skoro maja być wszystkie te submoduły podłączone do tego samego modułu to należałoby zdefiniować dla każdego z nich połączenie. Przydatny staje się wtedy wektor bram.

Zasady wykorzystania i deklaracji wektorów modułu i wektorów bram przedstawione zostały na przykładzie sieci klient-serwer. Nie będzie to sieć odwzorowująca rzeczywiste sieci jedynie będzie je przypominała układem czynników biorących udział w połączeniach i będzie służyła jako model edukacyjny.

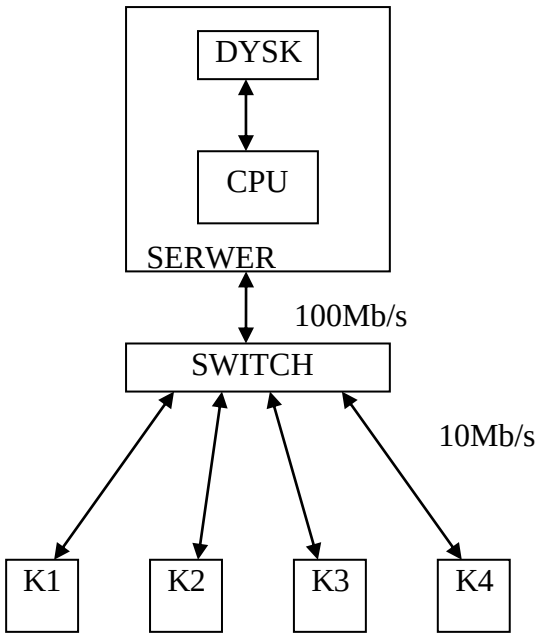
Przykład

Wykonany został model sieci komputerowej, w skład której wchodzą:

- serwer bazodanowy działający na takich samych zasadach jak w modelu z Lekcji 2(bez cache)
- cztery komputery klienckie (również działające tak jak w modelu z *Lekcji 2*)
- przełącznik switch pracujący z prędkością 10Mb/s na wejściach oraz 100Mb/s na wyjściach. Wewnętrzna przepustowość jest nieograniczona.

Klienci mają wysyłać żądanie do serwera poprzez przełącznik i otrzymywać odpowiedź. Schemat sieci przedstawiony jest na poniższym rysunku.

Dodatkowo zostaną wykorzystane (tak jak w poprzedniej lekcji) moduły sieciowe, które będą sprawdzały zajętość kanału transmisyjnego i pozwolą na bezkolizyjne wysyłanie komunikatów.



Rys. 1. Schemat sieci
Źródło: wykonanie własne

Z powyższego schematu łatwo można wywnioskować, że do budowy modelu będą potrzebne 3 moduły złożone:

- moduł `Serwer` – w skład którego wchodzą podmoduł procesor i dysk,
- moduł `Switch` – w którym zdefiniujemy podmoduł `Porty` zajmujący się sprawdzaniem skąd przybył komunikat i przesyłaniem go do odpowiedniej bramy oraz podmoduł `Sterowanie` zajmujący się ustawianiem odpowiednich rodzajów komunikatów co ułatwi rozpoznawanie dokąd `Porty` mają wysłać wiadomość,
- moduł `Symulator` – zawierający wymienione wyżej moduły i moduł `Klienta`.

Dla przejrzystości kodu oraz łatwiejszego odnajdywania ewentualnych błędów wszystkie moduły złożone można zapisać w osobnych plikach .ned i importować odpowiednie pliki (*serwer.ned* oraz *switch.ned*) do pliku, w którym zdefiniowany jest moduł nadrzędny (`Symulator`).

Zaczniemy od modułu serwera, który różni się od modułu `Serwer` z zadania z *Lekcji 2* jedynie tym, że jest nieco uproszczony - nie posiada podmodułu `Cache`. Zapisany zostanie on w pliku o nazwie *serwer.ned*.

Następnie omówiony zostanie moduł `Switch`.
Składa się on z submodułu odpowiadającego za przetrzymywanie informacji na temat przychodzących wiadomości, który został nazwany `Magazyn` oraz submodułu zajmującego się ustawianiem odpowiednich ~~kind'ów~~ adresów docelowych dla zadań – `Sterowanie_switchem`.

Do switcha mają być podłączeni klienci oraz serwer.
Wiadomości od klientów trafiają najpierw do `Magazynu`, następnie do `Sterowanie_switchem`, gdzie jest ustawiany odpowiedni adres docelowy (`destAddress`) i są z powrotem przesyłane do `Magazynu`. `Magazyn` wysyła komunikaty do `Serwera`.
Wynika stąd, że w `Magazynie` potrzebne są 4 połączenia odbierające wiadomości od klientów, 4 wysyłające do klientów, połączenie do i od sterowania oraz połączenie do i od serwera.

Wiedząc, że każdy klient wykonuje te same czynności można wykorzystać wektor bram, który wyznaczany jest za pomocą pary pustych nawiasów []. Tworzymy pliki: Magazyn.ned i Sterowanie_switchem.ned

```
//---------------- plik Magazyn.ned --------
simple Magazyn
{
    parameters:
        int czas_wyszukiwania_danych;
    gates:
        input od_klienta[];//wektor bram wejściowych od
klientów
        output do_klienta[];
        input od_sterowania[];
        output do_sterowania[];
        input od_cpu;
        output do_cpu;
}
//---------------- plik Sterowanie_switchem.ned --------
simple Sterowanie_switchem
{
    gates:
        input od_portow;
        output do_portow;
}
```

Wielkość wektora (czyli ilość bram w wektorze) określa się za pomocą `gates` (w poprzedniej wersji `gatesizes`) w module nadrzędnym w sekcji `submodules`. W tym przykładzie wykorzystana zostanie zmienna `ilosc_klientow`, która będzie przechowywała liczbę wszystkich klientów (jej wartość określona została w *omnetpp.ini*; elementy wektora numerowane są od 0). Jeśli pominięte zostanie nadanie rozmiaru wektora to zostanie on utworzony z wartością 0.

Należy pamiętać również, że komunikaty trafiają najpierw do modułu złożonego skąd są kierowane na `Magazyn`, więc konieczne jest zdefiniowanie wektora bram również w module `Switch`.

```
module Switch
{
    parameters:
        int id_klienta;
    gates:
        input do_switcha[];
        output od_switcha[];
        input od_serwera;
        output do_serwera;

    submodules:
        porty: Magazyn {
            @display("p=50,158");
            gates:
                od_klienta[id_klienta];
                do_klienta[id_klienta];
                od_sterowania;
                do_sterowania;
                od_cpu;
                do_cpu;
        }
        sterowanie: Sterowanie_switchem {
            @display(„p=50,50");
            gates:
                input od_portow;
                output do_portow;
        }
    connections:
      …
}
```

Do określenia odpowiednich połączeń pomiędzy wektorem bram `Switcha` a bramami klientów zastosowana jest pętla `for .. {}`.

```
connections:
        for i=0..(id_klienta-1) {
            do_switcha[i] --> porty.od_klienta[i];
            od_switcha[i] <-- porty.do_klienta[i];
               }
        od_serwera --> porty.od_cpu;
        do_serwera <-- porty.do_cpu;

        porty.od_sterowania <-- sterowanie.do_portow;
        porty.do_sterowania --> sterowanie.od_portow;
```

Następnie utworzony zostanie moduł złożony symulacji *Lekcja4.* W zadaniu potrzebnych jest czterech klientów, więc w sekcji `submodule` nadany jest wektor modułu `Klient`, a w sekcji `conections` za pomocą pętli `for` ustawione połączenia dla każdego modułu wektora.

Należy pamiętać również o określeniu odpowiedniej przepustowości na wejściu i wyjściu do switcha. Można to zrobić w dwa sposoby: „na szybko” i „elegancko”. Poniżej oba sposoby zostały przedstawione (`{ datarate = 10Mbps; }` oraz deklaracja typu `kanal_100 )`.

```
network Lekcja4
{
    parameters:
        int ilosc_klientow;
        @display(“bgb=930,500”);
    types:
        channel kanal_100 extends ned.DatarateChannel
        {
            datarate = 100Mbps;
        }

    submodules:
        host[ilosc_klientow]: Klient {
            parameters:
               @display(“p=105,420,row,200”);
        }
        siec_od[ilosc_klientow]: Siec {
            @display("p=65,320,row,200;i=block/ifcard");
        }

        siec_do[ilosc_klientow]: Siec {
            @display("p=165,320,row,200;i=block/ifcard");
        }
        switch: Switch {
            parameters:
               id_klienta = ilosc_klientow;
               @display(“p=420,225;i=device/switch”);
            gates:
                od_switcha[ilosc_klientow];
                do_switcha[ilosc_klientow];
                od_serwera;
                do_serwera;
        }
        komputer: Serwer {
            @display(„p=420,40;i=device/server”);
            gates:
                od_komputera;
                do_komputera;
        }
        sieciowa1: Siec {
            @display("p=350,120;i=block/ifcard");
        }

        sieciowa2: Siec {
```

```
            @display("p=475,120;i=block/ifcard");
        }
    connections:
        for i=0..ilosc_klientow-1 {
            siec_do[i].wyjscie --> {  datarate = 10Mbps; } -->
switch.do_switcha[i];
            siec_od[i].wyjscie --> {  datarate = 10Mbps; } -->
host[i].do_klienta;
            switch.od_switcha[i] --> siec_od[i].wejscie;
            siec_do[i].wejscie <-- host[i].od_klienta;

        }

        sieciowa2.wyjscie --> kanal_100 -->
serwer.do_komputera; //datarate = 100Mbps
        serwer.od_komputera --> sieciowa1.wejscie;
        switch.do_serwera --> sieciowa2.wejscie;
        sieciowa1.wyjscie --> kanal_100 --> switch.od_serwera;
}
```

Ponieważ moduł `Klient` jest wektorem modułów to podczas uruchomienia modelu w GNED widoczny będzie on jako jeden moduł. Dopiero w trakcie działania symulacji w trybie graficznym Tkenv pokazane zostaną wszystkie moduły wchodzące w skład wektora. Należy jednak ustawić ich położenie za pomocą odpowiedniej etykiety w `display`: **p=**`xpos,ypos,`**row**`,deltay,` dzięki której moduły ułożą się w jednym wierszu w odstępach od siebie określonych jako `deltay`.

Display posiada wiele różnych etykiet, które są przydatne przy ustawianiu modułów, m.in.:

- **p=**`xpos,ypos,`**column**`,deltay` – ustawia w kolumnie
- **p=**`xpos,ypos,`**ring**`,width,height` – ustawia w koło
- **b=**`width,height,`**rect** – ustawia w prostokąt

Przydatne jest również nadanie zmiennym wartości w pliku *omnetpp.ini*.

Model sieci jest już gotowy. Można go obejrzeć pod zakładką Design.

Teraz kolej jest na zaimplementowanie poszczególnych submodułów w C++.

Definicje wszystkich klas klasy `cSimpleModule` zostały zapisane w jednym pliku źródłowym **.h.* Konstrukcja obiektów klasy `cSimpleModule` została omówiona w *lekcji 2*. Dla przypomnienia poniżej znajduje się definicja klasy `Klient`.

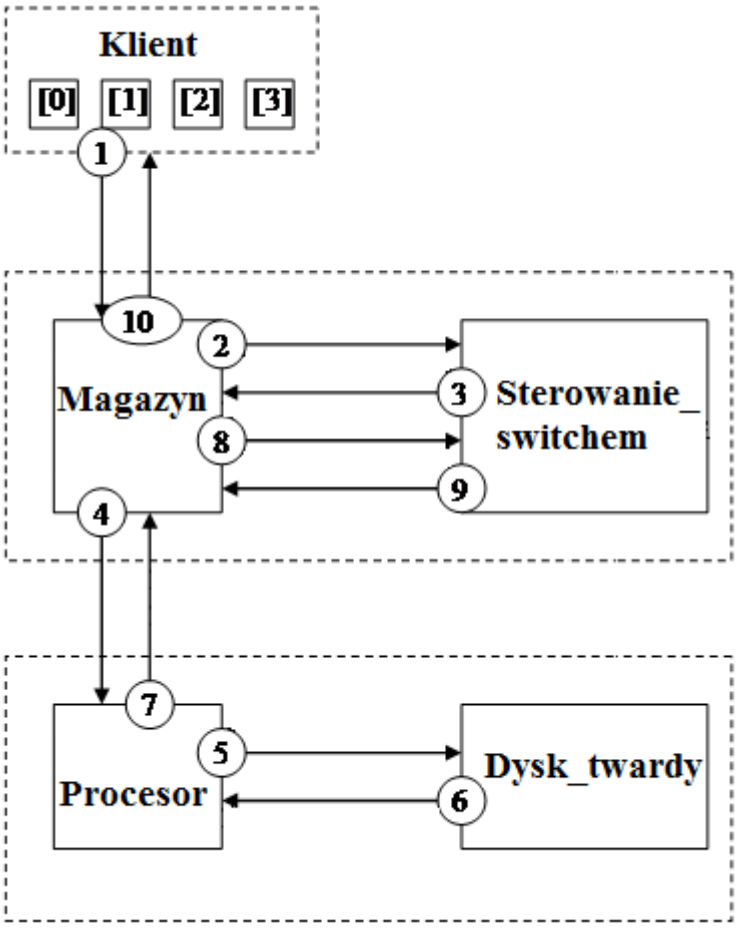
```
#ifndef __KLIENT_H__
#define __KLIENT_H__
```

```
#include <omnetpp.h>
#include "MyPacket_m.h"

class Klient : public cSimpleModule
{
     public:
          Klient() : cSimpleModule(32768) {}
          cQueue kolejka;
          MyPacket *zadanie;
          virtual void activity();
};
#endif
```

Podobnie jak w lekcjach wcześniejszych zastosowana została funkcja `activity()`.

Kierunek wysyłania wiadomości najłatwiej jest kontrolować operując na adresie docelowym komunikatu. Jednakże należy ustalić na samym początku algorytm przydzielania adresów dla poszczególnych wysłań wiadomości.
Poniższy rysunek przedstawia nadanie adresów w tym zadaniu.



Rys. 2. Nadawanie wiadomościom adresu docelowego (destAddress)
Źródło: wykonanie własne

Funkcja activity() modułu Klient ma na celu wysyłanie komunikatów do Switch'a. Ponieważ do poprawnego działania symulacji potrzebna jest informacja o tym, z którego modułu wektora została wysłana wiadomość aby wróciła do niego odpowiedź, należy z poszczególnych modułów wysyłać wiadomość zawsze o tym samym adresie źródłowym (srcAddress).

Brama wyjściowa w każdym module wektora ma tą samą nazwę, więc nie ma możliwości sugerowania się nią, ale każda brama wyjściowa modułu wektora jest połączona do innej bramy z wektora bram modułu sąsiedniego, czyli Magazyn. Można to sprawdzić za pomocą gate(„nazwa_bramy_wyjsciowej")->toGate(). Funkcja toGate() w Omnet 4.2 nazywa się getNextGate().Jest to dopiero nazwa tej bramy, a do zadania potrzebna jest jej wartość. Można ją uzyskać poprzez getIndex().

Niestety sposób ten nie zadziała w przypadku dołączenia do modelu - pomiędzy moduły Klient i Switch - modułów Siec. Wtedy najlepiej skorzystać z :
gate(„nazwa_bramy_wyjsciowej")->getOwnerModule()->getindex().

```
#include "Klient.h"
#include "MyPacket_m.h"

Define_Module(Klient);

void Klient::activity()
{
    kolejka.setName(„kolejka");

    for (;;)
    {
          char nazwa[20];
          int wielkosc_danych = par („wielkosc_danych");;
          strcpy(nazwa, „od~~do~~_klienta");
          MyPacket *zadanie = new MyPacket (nazwa);


         int id_klienta = gate("od_klienta")->getOwnerModule()
-> getIndex()+1;

          zadanie->setKind(id_klienta);
          zadanie->setSrcAddress(id_klienta);
          zadanie->setDestAdress(1);
          zadanie->setByteLength(wielkosc_danych);
          ev << „Klient do switcha zadanie z kind=" <<
```

```
zadanie->getKind() << „ ,Wielkosc danych= „ << wielkosc_danych
<< endl;

          send(zadanie, "od_klienta");
          double czas_namyslu=par („czas_namyslu”);

         waitAndEnqueue((double) czas_namyslu, &kolejka);
//tworzy kolejke komunikatow do tej kolejki
          kolejka.clear ();

    }
}
```

Wiadomość o odpowiednim adresie wysłana przez moduł `Klient` trafia do modułu `Siec`, a następnie do modułu `Magazyn`.
Sprawdzana jest kolejka i w przypadku, gdy jest pusta za zadanie podstawiona jest przybyła wiadomość, a w przeciwnym wypadku wiadomość z końca kolejki. Następnie sprawdzony jest adres docelowy tej wiadomości i na jego podstawie wybrana jest odpowiednia funkcja.

```
#include “Magazyn.h”

Define_Module(Magazyn);

void Magazyn::activity()
{
     MyPacket *zadanie;

     kolejka.setName(„kolejka”);

     for (;;)
     {
         if (kolejka.empty())
             zadanie = (MyPacket *) receive();
         else
       // w przeciwnym wypadku pobiera zadanie z konca kolejki
             zadanie = (MyPacket *)kolejka.pop();


         int dest = zadanie->getDestAddress();
         zadanie->setDestAddress(dest+1);

         double czas_wyszukiwania_danych =
         (double)par("czas_wyszukiwania_danych");

      //spr czy przyszlo od klienta
     if (dest == 1)
     {
     waitAndEnqueue(czas_wyszukiwania_danych, &kolejka);
     ev << „Magazyn dostal od klienta zadanie z kind=”
```

```
      << zadanie->getKind() << endl;
      wyslij_na_sterowanie(zadanie);
     }
     //spr czy przyszlo od sterowania i ma isc do serwer
     else if (dest == 3)
     {
     waitAndEnqueue(czas_wyszukiwania_danych, &kolejka);
     ev << „Magazyn dostal od sterowania zadanie z kind=”
     << zadanie->getKind() << endl;
          wyslij_do_serwera(zadanie);
     }
     //spr czy przyszlo od serwera
    else if (dest == 7)
    {
     waitAndEnqueue(czas_wyszukiwania_danych, &kolejka);
     ev << „Magazyn dostal od serwera zadanie z kind=”
     << zadanie->getKind() << endl;
     wyslij_na_sterowanie(zadanie);
     }
     //spr czy ma isc do klienta
     else if (dest == 9)
    {
     waitAndEnqueue(czas_wyszukiwania_danych, &kolejka);
     ev << „Magazyn dostal od sterowania zadanie z kind=”
     << zadanie->getKind() << endl;
     wyslij_do_klientow(zadanie);
     }
   }
}
```

Jeśli wiadomość przyszła od `Klienta` lub od `Serwera` to wiadomość wysłana jest do `Sterowania_switchem`.

```
void Magazyn::wyslij_na_sterowanie( MyPacket *zadanie)
{
     send(zadanie, "do_sterowania");
     ev << „Magazyn do sterowania zadanie z kind=” << zadanie-
>getKind() << endl;
}
```

Tak samo należy postąpić w przypadku zadań, które mają być wysłane do serwera.

```
void Magazyn::wyslij_do_serwera(MyPacket *zadanie)
{
     send (zadanie, "do_cpu");
     ev << „Magazyn do serwera zadanie z kind=” << zadanie-
```

```
>getKind() << endl;
}
```

Jeśli do `Magazynu` przybędzie wiadomość o adresie docelowym 9 to wiadomo że zadnie było już na serwerze i jest to odpowiedź do klienta.
W metodzie `send()` należy zaznaczyć przez którą bramę wektora ma być wysłana wiadomość.

```
void Magazyn::wyslij_do_klientow( cMessage *zadanie)
{
     int brama = zadanie->getSrcAddress();
     send (zadanie, „oddo_klienta$o”,brama-1);
     ev << „Magazyn do Klient zadanie z kind=” << zadanie-
>getKind() << endl;
}
```

Moduł `Sterowanie_switchem` ma bardzo proste działanie – zwiększa adres docelowy o 1 i wysyła z powrotem do `Magazynu`.

```
#include “Sterowanie_switchem.h”

Define_Module(Sterowanie_switchem);

void Sterowanie_switchem::activity()
{
  for (;;)
    {
     zadanie = (MyPacket *) receive();
      ev << „Sterowanie dostalo od Magazynu zadanie z kind=”
      <<zadanie->getKind()  << endl;
     wyslij_na_porty(zadanie);
    }
}

void Sterowanie_switchem::wyslij_na_porty(MyPacket *zadanie)
{

   int dest = zadanie->getDestAddress();
   zadanie->setDestAddress(dest+1);
   send (zadanie, "do_portow");

    ev << „Sterowanie do Magazyn zadanie z kind=” << zadanie-
       >getKind() << endl;
}
```

W module `Procesor` należy sprawdzić czy zadanie przyszło od `Switch’a` czy wróciło od dysku wykonać odpowiednią funkcję wysyłającą.

```
#include “Procesor.h”
Define_Module(Procesor);
```

```
void Procesor::activity()
{
     kolejka.setName("kolejka");

    for (;;)
    {
          int dest;

// jesli kolejka jest pusta to odbiera zadanie
//za pomoca funkcji receive

          if (kolejka.empty())
               zadanie = (MyPacket *) receive();
          else // w przeciwnym wypadku pobiera zadanie z konca kolejki
               zadanie = (MyPacket *)kolejka.pop();

        dest = zadanie->getDestAddress();
        zadanie->setDestAddress(dest+1);

          //spr czy przyszlo od switcha
          if (dest == 4)
          {
            ev << „Procesor dostal od switcha zadanie z kind="
            << zadanie->getKind() << endl;
            wysylamy_na_dysk(zadanie);
          }
        //spr czy wrocilo z dysku
          else
          {
               ev << „Procesor dostal od dysku zadanie z
           kind=" << zadanie->getKind() << endl;
               wysyla_do_switcha_od_serwera(zadanie);
          }
     }
}
void Procesor::wysylamy_na_dysk(MyPacket *zadanie)
{
     double czas_wyszukiwania_danych = (double)
       par(„czas_wyszukiwania_danych") ;
     waitAndEnqueue(czas_wyszukiwania_danych, &kolejka);

     ev << „Procesor do dysku zadanie z kind=" << zadanie-
        >getKind() << endl;
     send (zadanie, "do_dysku");
}

void Procesor::wysyla_do_switcha_od_serwera(MyPacket *zadanie)
{
     double czas_transmisji_do_switcha;
     czas_transmisji_do_switcha= (double) par
        („czas_transmisji_do_klienta");
```

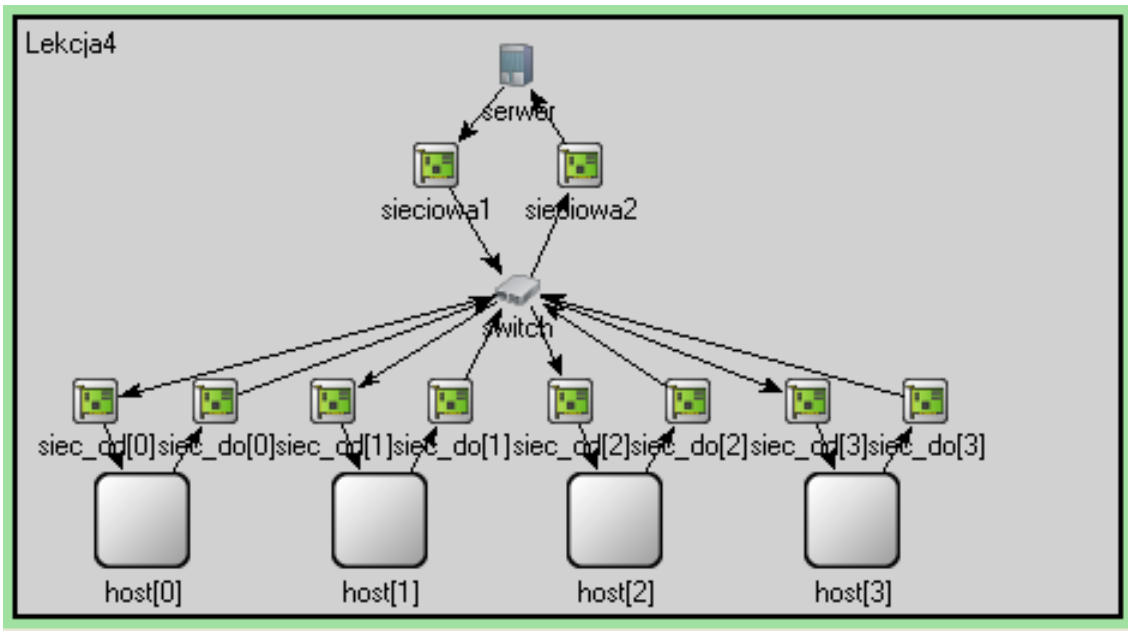
```
      czas_transmisji_do_switcha*= (double) (zadanie->
         getKind()*2);
      if (czas_transmisji_do_switcha>0 )
           waitAndEnqueue(czas_transmisji_do_switcha,
         &kolejka);

      ev << „Procesor do switcha zadanie z kind=” << zadanie-
         >getKind() << endl;
      send (zadanie, "do_klienta");
}
```

Jeśli zadanie przyszło od `switcha` należy zwiększyć adres docelowy i wysyłać wiadomość do modułu `Dysk_twardy`. Natomiast w przypadku wiadomości, która wróciła od dysku należy również zmienić adres docelowy i wysłać do `switcha`. Wykonane jest to na tej samej zasadzie jak w przypadków wcześniej już omawianych modułów.

Funkcje modułu `Dysk_twardy` nie wymagają sprawdzenia rodzaju wiadomości ponieważ dysk jest połączony jedynie z procesorem. Zadanie jest obsłużone na dysku, zostaje ustalony odpowiedni adres docelowy i zadanie wysłane jest do `Procesora`.

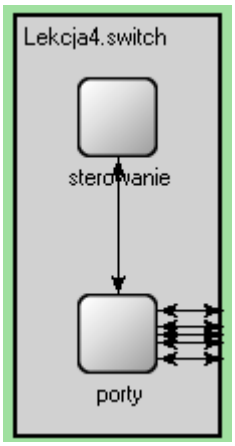
Zadanie jest już gotowe do uruchomienia i przeprowadzenia symulacji.



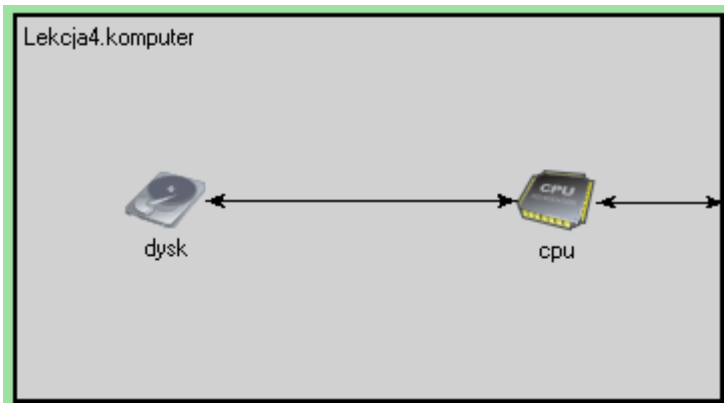
Rys. 3. Okno modułu Symulator
Źródło: wykonanie własne

Po uruchomieniu wyświetlone zostało okno modułu głównego. Klienci ustawieni są w szeregu w odstępach od siebie nadanych wcześniej w modelu modułu i podpisani są odpowiednia wartością wektora.

Po dwukrotnym kliknięciu na moduł switch oraz serwer otworzą się okna tych modułów.



Rys. 4. Okno modułu switch
Źródło: wykonanie własne



Rys. 5. Okno modułu serwer
Źródło: wykonanie własne

Zadanie do samodzielnego przygotowania:

Wykonaj model sieci komputerowej, w skład której wchodzą: określona liczba (podana przy rozpoczęciu symulacji) serwerów bazodanowych (z zadania drugiego), pewna liczba klientów (podana przy rozpoczęciu symulacji), przełącznik sieciowy (switch), sieć pracującej z prędkością 10 Mb/s. Wewnętrzna przepustowość przełącznika wynosi 100 Mb/s. Klient powinien wysyłać żądanie do wylosowanego wcześniej serwera.