Lekcja 2. Tworzenie i uruchamianie prostej symulacji

Lekcja druga poświęcona jest wyjaśnieniu budowy symulatora, implementacji modułów w C++ i ich kompilacji. Objaśnione zostaną potrzebne do symulacji funkcje, plik omnetpp.ini oraz krok po kroku przedstawione zostanie uruchamianie symulacji i obsługa interfejsów użytkownika.

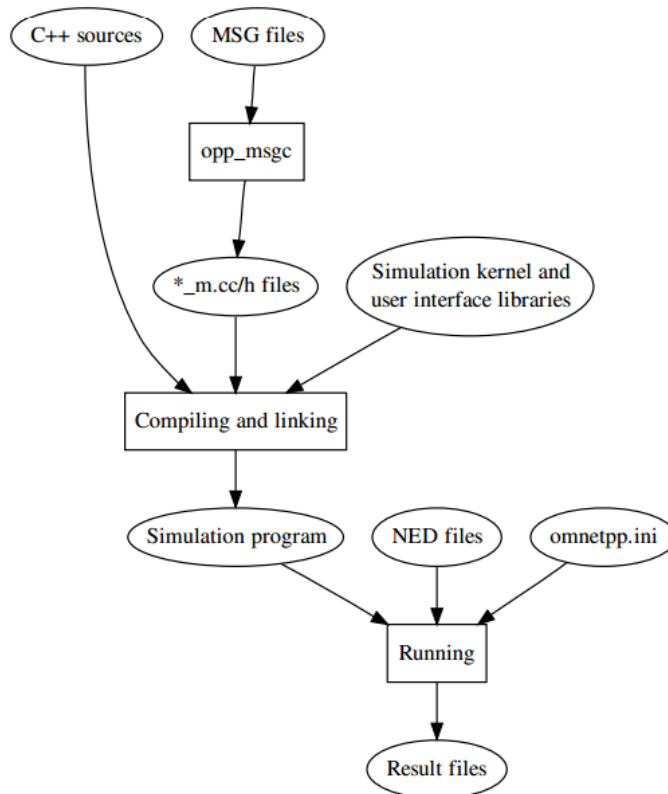
Budowa symulatora i uruchamiania symulacji

Jak już było wspomniane w *lekcji 1* do stworzenia kompletnego modelu symulacyjnego w OMNeT++ potrzebne są następujące elementy:

- opis topologii modelu NED (pliki z rozszerzeniem *.ned*)
- plik sterujący – *omnetpp.ini*
- implementacje modułów prostych w języku C++ (pliki z rozszerzeniem .*cpp* oraz .*h*)
- definicje przesyłanych komunikatów (jeżeli są wykorzystywane) (pliki z rozszerzeniem .*msg*)

Przed uruchomieniem symulacji pliki **.ned* i **.msg* muszą zostać skompilowane do postaci kodu w C++ za pomocą dostarczonych przez OMNeT++ kompilatorów, a następnie wszystkie pliki **.cpp* i **.h* są kompilowane i linkowane z dostarczoną biblioteką symulacyjną (kernelem - jądrem symulacji) i interfejsem użytkownika (ułatwia debugowanie, demonstracje oraz wsadowe uruchamianie symulacji).

Proces budowy i uruchamiania symulacji najlepiej obrazuje poniższy schemat



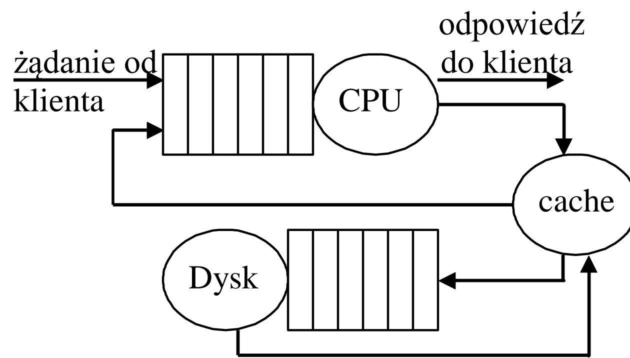
Rys. 1. Schemat budowy i uruchamiania symulacji
źródło: OMNeT++ User Manual v.4.2

OMNeT ++ ma obszerną bibliotekę klas C++, której można używać do implementacji modułów prostych. Posiadają one różne metody wykorzystywane do efektywnego modelowania symulacji:

- wysyłanie i odbieranie komunikatów: `cMessage, cPacket`
- dostęp do bram i parametrów modułu: `cGate`
- uzyskanie dostępu do innych modułów w sieci
- generowanie liczb losowych
- dostęp do parametrów modułu: `cPar`
- przechowywanie danych: `cQueue`
- zapisywanie statystyk do plików: `cOutVector`
- zbieranie prostych statystyk: `cStdDev` i `cWeightedStddev`
- szacowanie rozkładów: `cLongHistogram, cDoubleHistogram, cVarHistogram, cPSquare, cKSplit`

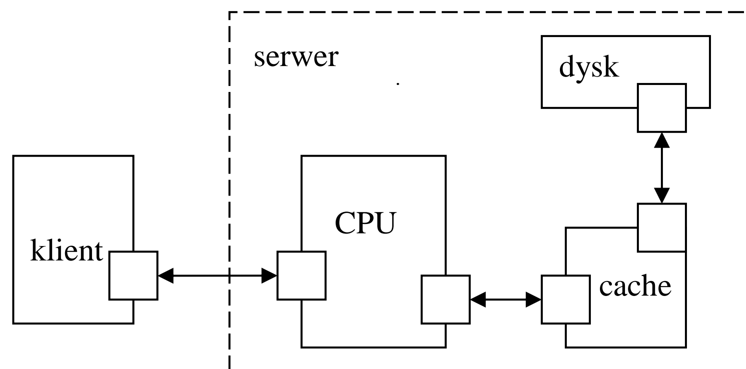
Przykład

Struktura poszczególnych plików symulacyjnych zostanie wyjaśniona na podstawie symulatora serwera bazodanowego (schemat sieci kolejkowej na rysunku 2).



Rys. 2. Schemat sieci kolejkowej
źródło: wykonanie własne

Wykonany zostanie model symulacyjny klienta wysyłającego żądania do serwera, w skład którego wchodzi procesor, dysk oraz pamięć podręczna (cache). Schemat połączeń pomiędzy poszczególnymi elementami został przedstawiony na rysunku 3.



Rys. 3. Schemat połączeń
źródło: wykonanie własne

Częstotliwość, z jaką klienci będą wysyłali żądania jest rozkładem wykładniczym o wartości oczekiwanej 0,02s, a wielkość danych, które będą żądali jest rozkładem normalnym z wartością oczekiwaną μ=10000B i wariancją σ2=3333B. Przyjmiemy także, że żądanie trafia w pierwszej kolejności do procesora. Po otrzymaniu żądania demon serwera wykonujący obsługę żądania zużywa 527 ms na procesorze w celu przygotowania odpowiedzi.

Jeśli dane nie znajdują się w pamięci podręcznej serwera demon pobiera dane z dysku (dla 80% żądań dane znajdują się w pamięci podręcznej). Natomiast jeżeli dane znajdują się w pamięci podręcznej, żądanie nie jest wysyłane do dysku, tylko powrotnie przesłane do kolejki procesora i odpowiedź do klienta.
Dodatkowo przyjmijmy, że dysk ma być przeszukiwany w celu znalezienia danych w czasie 28ms, czas transmisji ma wynosić 410 μs na każde 4KB. Uzyskane dane w drodze powrotnej trafiają do pamięci podręcznej, a dalej do procesora. Po odnalezieniu danych procesor przesyła dane do klienta (koszt

transmisji na procesorze to 240 μs na każde 512B).
Sieć, którą będzie połączony klient z serwerem ma nieskończoną przepustowość.

Plik .ned

Pliki **.ned* były omówione w *lekcji 1* i przygotowaliśmy już częściowy model topologii sieci dla tego zadania.
Potrzebujemy jedynie uzupełnić go o podmoduł typu cache (i związane z nim bramy) oraz parametry, które potrzebujemy do zadania.

```
simple Cache
{
    parameters:
        int procent_danych_w_pamieci;
    gates:
        inout oddo_procesora;
        inout oddo_dysku;
}
```

Powyższe bramy należy uwzględnić w `Symulatorze`.

Do poszczególnych modułów dopiszemy parametry wymagane do poprawnego działania programu, czyli wielkość przekazywanych żądań, czas wyszukiwania, czas transmisji, itd.

```
simple Klient
{
    parameters:
       volatile int wielkosc_danych;
       volatile int czas_namyslu;
    gates:
        //
}


simple Dysk_twardy
{
    parameters:
        int czas_przeszukiwania_i_oczekiwania @unit("us");
        int czas_transmisji @unit("us");
    gates:
        //
}




simple Procesor
```

```
{
    parameters:
        int czas_wyszukiwania_danych @unit("us");
        int czas_transmisji_do_klienta @unit("us");
    gates:
        //
}
```

W sekcji `connections` można ustawić odpowiednią przepustowość danych (`datarate`) czyli szerokość pasma kanału. Przepustowość danych wyrażana jest w bitach na sekundę i jest używana do obliczenia czasu transmisji pakietu. Ponieważ mamy zaznaczone w zadaniu, że przepustowość między klientem a serwerem jest nieskończona, a taka właśnie jest domyślna, więc nie musimy zaznaczać tego w kodzie.

```
network Symulator
{
    @display("bgb=414,330");
    submodules:
        Host: Klient {
            parameters:
                wielkosc_danych= lognormal(40,5,0);
                    czas_namyslu= exponential(0.5);
                @display("p=333,156;i=device/pc2");
        }
        Komputer: Serwer {
            @display("p=79,156;i=device/server2");
        }
    connections:
        Host.oddo_klienta <--> Komputer.oddo_komputera;
}
```

Pliki *.cpp i *.h

Mamy już gotowy model modułów, więc teraz możemy przejść do ich implementowania w C++.

W OMNeT++ to moduły proste odpowiadają obsługę zdarzeń zachodzących podczas symulacji.

Typy modułów prostych tworzone są za pomocą klasy `cSimpleModule`. W niej możemy zaimplementować następujące funkcje odpowiedzialne za odpowiednie działanie:

- `void handleMessage (cMessage *msg)`- funkcja, która będzie wykorzystywana dla każdej wiadomości, która przyjdzie do modułu (więcej informacji na temat tej funkcji opisane zostało w *Lekcji5*)
- `void initiialize()` – wykonuje wszystkie funkcje inicjalizacji (czyta

parametry modułu, wprowadza zmienne, alokuje dynamiczne struktury danych). Wywołana jest przed przetworzeniem pierwszego zdarzenia.
- `void activity()` – w niej definiujemy funkcje, dzięki którym można m.in. odbierać wiadomości (`receive()`), zawiesić działanie modułu na jakiś czas (`wait()`), wysłać wiadomość (`send()`),
- `void finish()` – rejestruje statystyki po zakończeniu symulacji. Wywołana jest tylko w przypadku poprawnego zakończenia symulacji (bez błędów). Nie posiada dostępu do zmiennych lokalnych funkcji `activity()`.

Zaczniemy od implementowania modułu Klient, który generuje żądania do procesora i ustawia je do kolejki.

Zarówno funkcja `handleMessage()` jak i `activity()` są używane podczas przetwarzania i wpływają na zachowanie modelu.

Wykorzystamy funkcje `activity()` oraz klasę `cQueue`, które zdefiniujemy w pliku *nagłówkowym .h*.

`CQueue` działa jako kolejka. Można przechowywać w niej obiekty typów np. `cMessage` czy `cPar`. Jej podstawowe funkcje to `insert(cObject *obj)` – służy do wprowadzania elementów do kolejki oraz `pop()`, która pobiera elementy z kolejki. Funkcja `empty()` informuje czy kolejka jest pusta. Możemy również sprawdzić liczbę obiektów w kolejce wykorzystując metodę `length()`.

```
class Klient : public cSimpleModule
{
  protected:
      virtual void initialize();
      virtual void handleMessage(cMessage *msg);
  public:
       Klient() : cSimpleModule(8192){}
       cQueue kolejka;

      virtual void activity();

};
```

Klasa `cSimpleModule` musi być zadeklarowana w OMNeT++ za pomocą komendy `Define_Module()`. Definicja modułu musi koniecznie znajdować się w pliku *.cpp.* W przypadku naszego zadania będzie to plik *klient.cpp*.

```
Define_Module(Klient);
```

Musimy również dołączyć plik *klient.h* i inne potrzebne pliki nagłówkowe za pomocą include.

```
#include "klient.h"
```

Możemy przejść do inicjowania funkcji `activity()`.
Funkcja `activity()` standardowo składa się z niekończącej się pętli. Możemy w niej wykorzystać m.in. metody takie jak `wait()`, `receive()` czy `send()`.

Wiemy, że nasz klient generuje zadania i wysyła je do kolejki serwera, więc musimy zadeklarować tą kolejkę.

```
void Klient::activity()
{
    kolejka.setName("kolejka");
    for (;;)
    {
     // . . .
    }
}
```

Jak już było wspomniane w *lekcji 1* moduły w OMNeT++ komunikują się ze sobą za pomocą wysyłanych komunikatów należących do klasy `cMessage`.
Nowe komunikaty tworzymy za pomocą operatora `new` i usuwamy przy pomocy `delete`. Odwołujemy się do nich przez wskaźnik (`cMessage *wskaznik`) oraz możemy nadać im nazwę (`new cMessage("nazwa”))`.

W pętli `for` zadeklarujemy, więc wskaźnik na wiadomość, którą chcemy wysłać do modułu `Serwer`.

```
char nazwa[20];

strcpy(nazwa, "oddo_klienta");
cPacket *zadanie = new cPacket(nazwa);
```

Za pomocą metody `par()` uzyskamy dostęp do parametrów modułu określonych w topologii modelu. Wykorzystamy również `setKind` do określenia rodzaju komunikatu. Będzie nam to przydatne do określania, od którego modułu przyszedł komunikat.

```
int wielkosc_danych;

wielkosc_danych=par ("wielkosc_danych");
zadanie->setByteLength(wielkosc_danych);
zadanie->setKind(1);

ev << "Klient: Wielkosc wygenerowanego zadania: "
   << wielkosc_danych << endl;
ev << "Klient: Rodzaj komuniaktu od klienta: " << zadanie->getKind
()
   << endl;
```

Dzięki ev możemy wyświetlać w oknie głównym informacje na temat danych

wyjściowych, co pozwoli nam śledzić wartości pojawiające się podczas symulacji.

Teraz możemy już wysłać wiadomość do serwera.

Wysyłanie wiadomości odbywa się przy pomocy funkcji `send(cMessage *msg, „nazwa_bramy”)`. Musimy również pamiętać o tym, że zadania mają trafiać do kolejki. Bramy dwukierunkowe mają dodatkowo oznaczenie rodzaju portu wejściowy lub wyjściowy („nazwa$i”, „nazwa$o”).

```
double czas_namyslu;

...
send(zadanie, "oddo_klienta$o");
czas_namyslu=par ("czas_namyslu");
waitAndEnqueue((double) czas_namyslu, &kolejka);

kolejka.clear ();
```

Kolej na zaimplementowanie podmodułów Serwera: `Procesor`, `Cache` oraz `Dysk_twardy`.

Pliki nagłówkowe tworzymy analogicznie jak dla *klient.h*: deklarujemy klasy `Procesor`, `Cache`, `Dysk_twardy` za pomocą klasy `cSimpleModule` oraz potrzebne do symulacji funkcje. Funkcja `handleMessage` jest wywoływana za każdym razem gdy przybywa wiadomość.
W zależności od bramki do której trafia wiadomość `handleMessage` wywołuje różne funkcje obsługi *Msg do daleszej obróbki wiadomości.
Dla procesora plik *procesor.h* będzie wyglądał następująco:

```
#ifndef __PROCESOR_H__
#define __PROCESOR_H__

#include <omnetpp.h>

class Procesor : public cSimpleModule {
     protected:
          virtual void initialize();
          virtual void handleMessage(cMessage *msg);
     public:
         Procesor() : cSimpleModule(32768) {}
         cQueue kolejka;
         cPacket *zadanie;

         virtual void activity();

         virtual void obsluz_zad_na_procesorze(cPacket *);
         virtual void wysyla_do_klient(cPacket *);
};
```

```
#endif
```

Pliki *cachek.h* oraz *dysk_twardy.h* będą różniły się oczywiście wykorzystanymi funkcjami:

```
class Cache : public cSimpleModule
{
  ……
  public:
      Cache() : cSimpleModule(32768){}
      ……
      virtual void activity();

      virtual void obsluz_zad_na_cache(cPacket *);
      virtual void wyslij_do_dysku(cPacket *);

};
```

```
class Dysk_twardy : public cSimpleModule
{
  ……
  public:
      Dysk_twardy() : cSimpleModule(32768){}
   ……

      virtual void activity();
      virtual void wysylanie_do_cache(cPacket *);
};
```

Następnie musimy w plikach **.cpp* zadeklarować klasy `cSimpleModule` za pomocą komendy `Define_Module(nazwa_modułu)`.

Ponieważ żądanie od klienta trafia najpierw do procesora, więc omówiony zostanie ten moduł jako pierwszy.

Funkcję `activity()` wykorzystamy do sprawdzenia czy w kolejce są już jakieś zadania do obsłużenia. Jeśli kolejka jest pusta to zadanie jest od razu obsłużone, natomiast w przeciwnym wypadku obsłużymy zadanie z kolejki. Do odbioru wiadomości w funkcji `activity()` służy funkcja `receive()`.

Za pomocą `kind()` musimy sprawdzić czy zadanie przyszło od klienta, czy od dysku i wywołać odpowiednią funkcję. Możemy to sprawdzić, ponieważ w każdym module ustawiamy rodzaj wiadomości za pomocą `setKind()`.

Wiadomość od klienta ustawiliśmy na 1, wiadomość od procesora do pamięci podręcznej ustawimy na 2, od cache do dysku na 3, od dysku do cache – 4, od cache do procesora na 5, a odpowiedź od procesora do klienta na 6.

Teraz nie mamy już problemu ze sprawdzeniem, który moduł przysłał zadanie.

```
Define_Module(Procesor);

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

    for (;;)
    {
     if (kolejka.empty())
          zadanie = (cPacket *) receive();
     else
         zadanie = (cPacket *)kolejka.pop();

     if (zadanie->getKind()==1)  //od klienta
          obsluz_zad_na_procesorze(zadanie);

     if (zadanie->getKind()==5)  //od pamięci podręcznej
          wysyla_do_klient(zadanie);
    }
}
```

Jeśli żądanie zostało przysłane od `Klienta` to musimy je obsłużyć i wysłać przez odpowiednią bramę do `Cache`.

```
void Procesor::obsluz_zad_na_procesorze(cPacket *zadanie)
{
     char nazwa [15];

     double czas_wyszukiwania_danych = (double)
            par("czas_wyszukiwania_danych") ;
     waitAndEnqueue(czas_wyszukiwania_danych, &kolejka);

     ev << "procesor: Wyslanie zadania do cache" << endl;

     zadanie->setKind(2);

     ev << "procesor: Rodzaj komuniaktu wsylanego do cache: "
        << zadanie->getKind () << endl;

     send (zadanie, "oddo_cache$o");
}
```

Funkcja ta nie wymaga wyjaśnień, ponieważ wykorzystaliśmy metody już wcześniej omawiane.

Podobnie jest w przypadku funkcji wysyłającej odpowiedź do klienta.

```
void Procesor::wysyla_do_klient(cPacket *zadanie)
```

```
{
     char nazwa [15];
     double czas_transmisji_do_klienta;

     czas_transmisji_do_klienta= (double)
           par ("czas_transmisji_do_klienta")*
                            ( zadanie-> getByteLength ()/512);

     if (czas_transmisji_do_klienta>0 )
        waitAndEnqueue(czas_transmisji_do_klienta, &kolejka);

     ev << "procesor: Wysylanie odpowiedzi do klienta" << endl;
          zadanie->setKind(6);
     send (zadanie, "oddo_klienta$o");
}
```

Po wysłaniu żądania do `Cache` musimy sprawdzić czy znajdują się tam poszukiwane dane (z treści zadania wiemy, że dla 80% żądań dane znajdują się w pamięci podręcznej). Będziemy losowali jakaś wartość z określonego zakresu liczb, w którym 80% liczb będzie odpowiadało znalezieniu danych.
Jeśli wystąpi liczba z tych 20% to wysłany zostanie komunikat do dysku i tam zostanie obsłużone żądanie, w przeciwnym wypadku wysłany zostanie komunikat powrotem do procesora.

```
Define_Module( Cache );

void Cache::activity()
{
     double procent_danych_w_pamieci;

     for (;;)
     {
     zadanie = (cPacket *) receive();

       //jeśli wiadomość przyszła od procesora
     if (zadanie->getKind()==2)
     {
          procent_danych_w_pamieci= rand () %10 +1;
          if (procent_danych_w_pamieci <= 8)
              obsluz_zad_na_cache(zadanie);
           else
              wyslij_do_dysku(zadanie);
     }
       //jęsli wiadomość przyszła od dysku
     if (zadanie->getKind()==4)
           obsluz_zad_na_cache(zadanie);
     }
}
```

Funkcje wysyłania komunikatów do dysku oraz procesora są analogiczne do funkcji występujących w module `Procesor`.

Pozostaje nam już tylko napisane pliku konfiguracyjnego.

Plik konfiguracyjny

Plik *omnetpp.ini* przeznaczony jest do umieszczania w nim wartości parametrów dla modułów oraz ustala się w nim przebieg całej symulacji.

Plik ten czytany jest przez program przy starcie symulacji i składa się z sekcji, których nazwy zapisuje się w kwadratowych nawiasach:

- `[General]` – w tej sekcji mogą znajdować się takie opcje jak dynamiczne ładowanie modułów (`preload-ned-files=*ned`), domyślna sieć dla uruchamianej symulacji (`network=nazwa_sieci`), limit czasu symulacji (`sim-time-limit = czas_symulacyjny`), limit rzeczywistego czasu działania programu (`CPU-time-limit=rzeczywisty_czas`), nazwy plików z danymi statystycznymi dla narzędzi scalars i plove (`output-vector-file = plik.vec, output-scalar-file = plik.sca`).
  W przypadku naszego zadania sekcja ta wyglądać, więc będzie tak:

```
[General]
network = Symulator
```

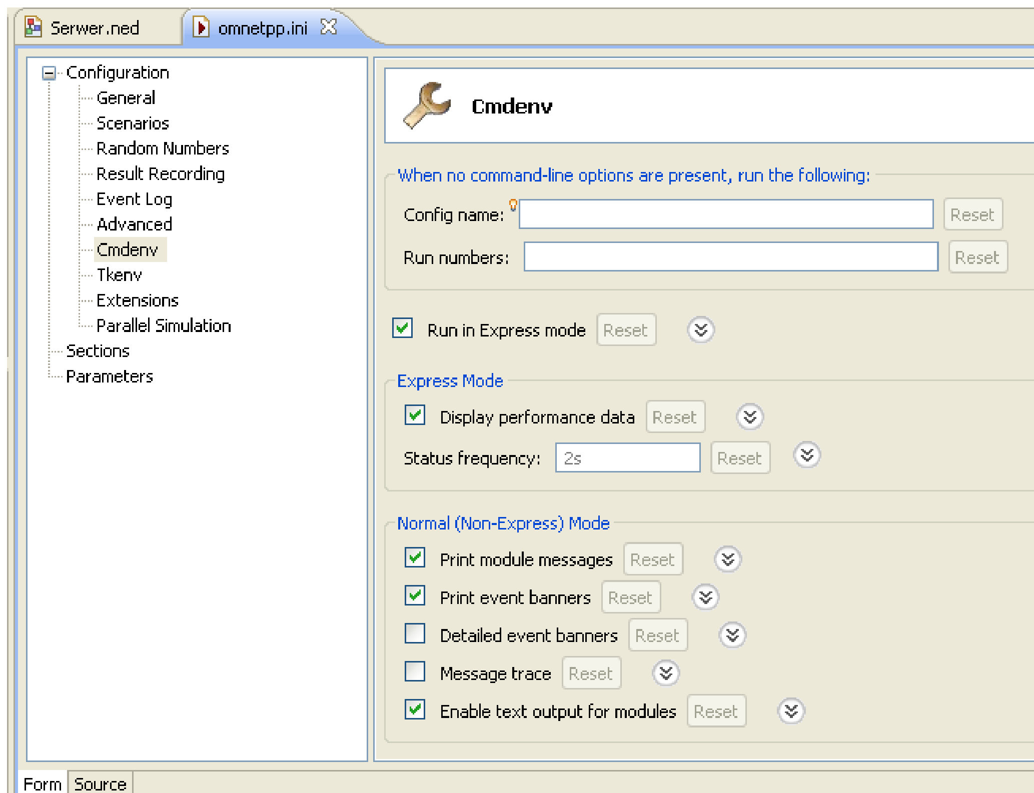
- [`Config nazwa`] – w tej sekcji można umieścić dowolnie nazwane konfiguracje symulacyjne. Przykładem takich danych mogą być zmienne wykorzystywane w symulacji:

```
[Config Lekcja2]

Lekcja2.Komputer.cpu.czas_wyszukiwania_danych= 0.000527
Lekcja2.Komputer.cpu.czas_transmisji_do_klienta=0.00024
Lekcja2.Komputer.dysk.czas_transmisji = 0.00041

Lekcja2.Klient1.wielkosc_danych = normal(10000,57.73,0)
Lekcja2.Klient1.czas_namyslu = exponential(0.02)
Lekcja2.Klient1.iden=1
```

W wersji 4.2 Omnet'a++ dostępny jest obok widoku „source" z zawartością pliku omnetpp.ini formularz, w którym zamiast wpisywania kolejnych linii kodu możemy wypełnić dany formularz konfiguracyjny. Widok tego formularza przedstawiony został na rys 4. Ponadto (w wersji 4.2)nie jest konieczna sekcja parameters, gdyż wszystkie parametry symulacyjne ujęte zostały w sekcjach parameters poszczególnych modułów (w plikach .ned). Większość parametrów ujętych jest w pliku serwer.ned

```
module Serwer
{
```

```
        @display("bgb=351,230");
        gates:
            inout oddo_komputera;
        submodules:
            cpu: Procesor {
                parameters:
                    czas_wyszukiwania_danych= 0.527us;
                    czas_transmisji_do_klienta= 0.00024us;
                    @display("p=269,91;i=device/cpu");
            }
            dysk: Dysk_twardy {
                parameters:
                    czas_transmisji = 14us;
                    czas_przeszukiwania_i_oczekiwania= 0.028us;;
                    @display("p=74,91;i=device/disk");
            }
            bufor: Cache {
                parameters:
                    procent_danych_w_pamieci= uniform(1, 10);
                    @display("p=174,91;i=block/boundedqueue");
            }
        connections:
            oddo_komputera <--> cpu.oddo_klienta;
            cpu.oddo_cache <--> bufor.oddo_procesora;
            bufor.oddo_dysku <--> dysk.oddo_cache;
}
```



Rys. 4. Formularz konfiguracyjny
źródło: wykonanie własne

Parametry losowe – rozkłady

Numeryczne parametry mogą zawierać losowe wartości o różnych rozkładach. Takie parametry zwracają różne wartości, za każdym razem, gdy są inicjowane, gdy są czytane z modułu prostego. Parametry takie mogą być deklarowane jako `const`, jeśli chcemy, aby wartość losowa została im przypisana tylko raz na początku.

Generatory liczb losowych (RNGs) mogą wykorzystywać rozkłady dostarczone przez OMNeT++.

Funkcja	Opis
Dystrybucje ciągłe	
`uniform (a, b, rng=0)`	Rozkład jednostajny w przedziale[a,b)
`exponential (mean, rng=0)`	Rozkład wykładniczy z nadanym parametrem mean
`normal (mean, stddev, rng=0)`	Rozkład normalny z nadanym parametrem mean i standardowym odchyleniem stddev
`truncnormal (mean ,stddev, rng=0)`	Rozkład normalny skrócony do wartości nieujemnych
`gamma_d (alpha, beta, rng=0)`	Rozkład gamma z parametrami alfa > 0 , beta >0
`beta (alphal, alpha2, rng=0)`	Rozkład beta z parametrami alfa1>0 , alfa2 >0
`Erlang_k (k, mean, rng=0)`	Rozkład erlang stopnia k>0 i nadaną wartością średnią
`chi_square (k, rng=0)`	Rozkład chi-square ze stopniem k>0
`student_t (i, rng=0)`	Rozkład student-t ze stopniem i>0
`Cauchy (a, b, rng=0)`	Rozkład cauchy z parametrami a,b gdzie b>0
`triang (a, b, c, rng=0)`	Rozkład trójkątny z paramertami a<=b<=c, a!=c
`longnormal (m, s, rng=0)`	Rozkład logarytmiczno-normalny z wartością średnią m i odchyleniem s>0
`weibull (a, b, rng=0)`	Rozkład weibull z parametrami a>0,b>0
`pareto_shifted (a, b, c, rng=0)`	Uogólniony rozkład Pareto z parametrami a, b i przesunięciem c
Dystrybucje nieciągłe	
`intuniform (a, b, rng=0)`	Jednolita liczba całkowita od a.,b
`bernoulli (p, mg=0)`	Rozkład próby Bernoulliego z

	prawdopodobieństwem 0<=p<=1
`binomial (n, p, rng=0)`	Rozkład binomial z parametrami n>=0 i prawdopodobieństwem 0<=p<=1
`geometric (p, rng=0)`	Rozkład geometryczny z prawdopodobieństwem 0<=p<=1
`negbinomial (n, p, rng=0)`	Rozkład binomial z parametrami n>0 i prawdopodobieństwem 0<=p<=1
`poisson (lambda, mg=0)`	Rozkład poisson z parametrem lambda

Jeśli nie wyszczególnimy opcjonalnego argumentu rng, funkcje użyją generatora liczb losowych 0.

Uruchamianie symulacji

Skoro mamy już gotowe wszystkie potrzebne do symulacji pliki możemy przystąpić do tworzenia pliku wykonywalnego.

Przed przystąpieniem do kompilacji należy skopiować do jednego folderu np. *Lekcja2* (nazwa folderu) wszystkie pliki modelu symulacyjnego: pliki **.ned* , pliki **.cpp*, pliki **.h* oraz plik sterujący *omnetpp.ini*.

Kompilację i symulację możemy uruchomić zarówno w konsoli (mingwenv.cmd) lub w środowisku graficznym.

Są dwa sposoby uruchomienia symulacji:

- w trybie linii poleceń – `Cmdenv`
- w trybie graficznym - `Tkenv`

Tryb linii poleceń nie posiada wizualizacji, więc informacje z przebiegu symulacji są jedynie drukowane na ekran lub zapisane do pliku.
W tym trybie mamy do wyboru tylko dwie możliwości uruchamiania:

- `normal` (non-expres) – wszystkie informacje są wypisywane na ekran w trakcie symulacji
- `express` – nadaje się do dłuższych symulacji, może wypisywać zbiorcze informacje tylko co pewien czas lub po zakończeniu symulacji.

Aby korzystać z tego trybu uruchamiania w pliku *omnetpp.ini* powinna znaleźć się sekcja [`Cmdenv`]. Jej podstawowe opcje to:

- *express-mode = yes* – przeznaczone dla długich symulacji, nie wypisuje komunikatów na ekran, domyślnie tryb jest ustawiony na "*normal-mode*"
- *module-messages = yes* - tylko w trybie *normal*, wypisuje na ekran wszystkie komunikaty z modułów
- *event-banners = yes* - tylko w trybie *normal*, wypisuje informacje o zajściu zdarzenia
- *status-frequency=50000* - tylko w trybie *express*, wypisuje stan symulacji co określoną liczbę zdarzeń

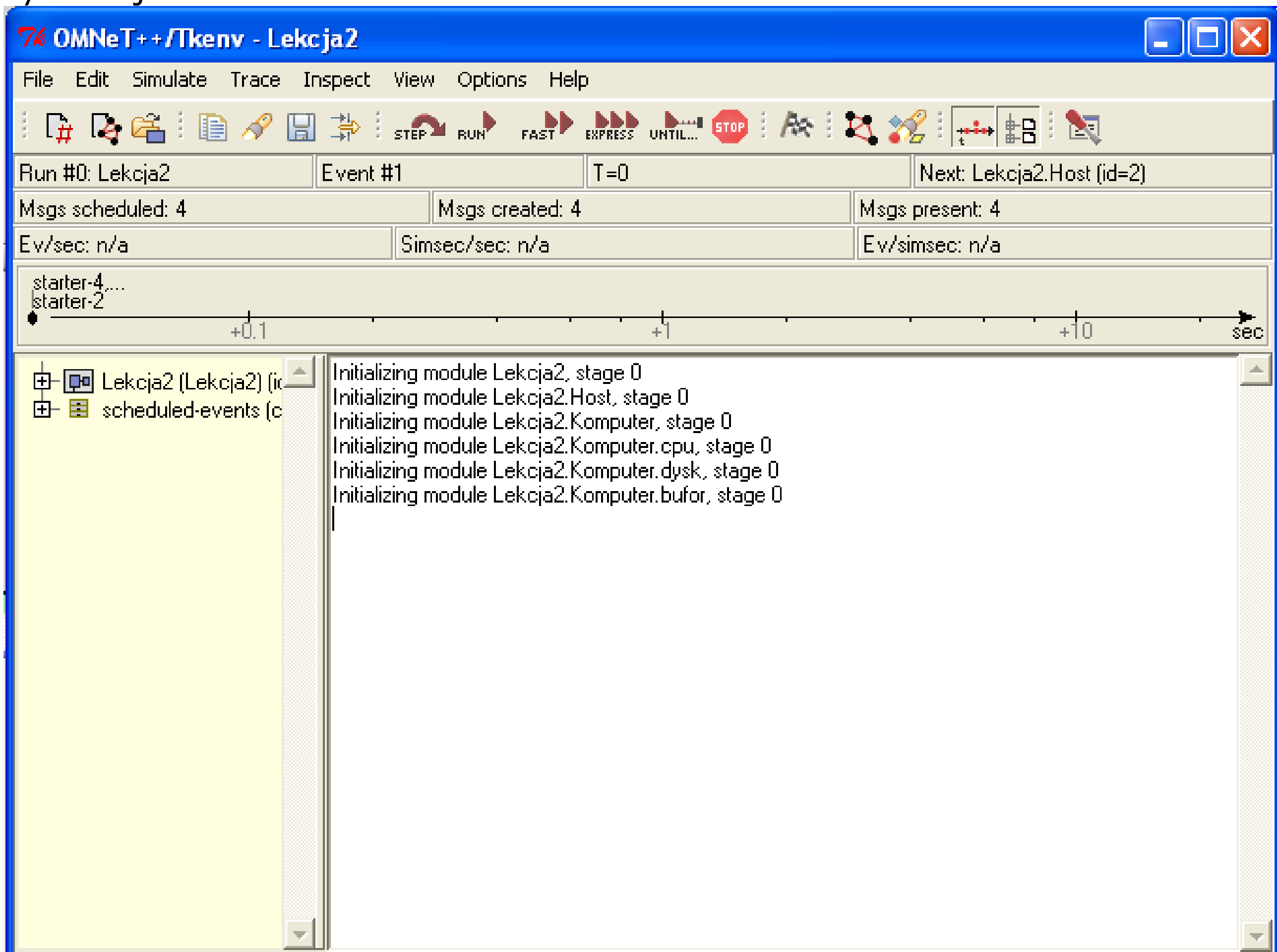
- *message-trace=no* - wypisuje informacje o każdym zdarzeniu wysyłania komunikatu (domyślnie wyłączone)

Interfejs `Tkenv` jest graficznym okienkowym interfejsem użytkownika. Korzystając z tego trybu jest możliwość szczegółowej wizualizacji przebiegu całej symulacji, oraz możliwość oglądania statystyk i wektorów wyjściowych podczas symulacji oraz po jes zakończeniu. Ze względu na możliwość przeglądania każdego modułu (także każdego w osobnym oknie) można z łatwością zmieniać im parametry.

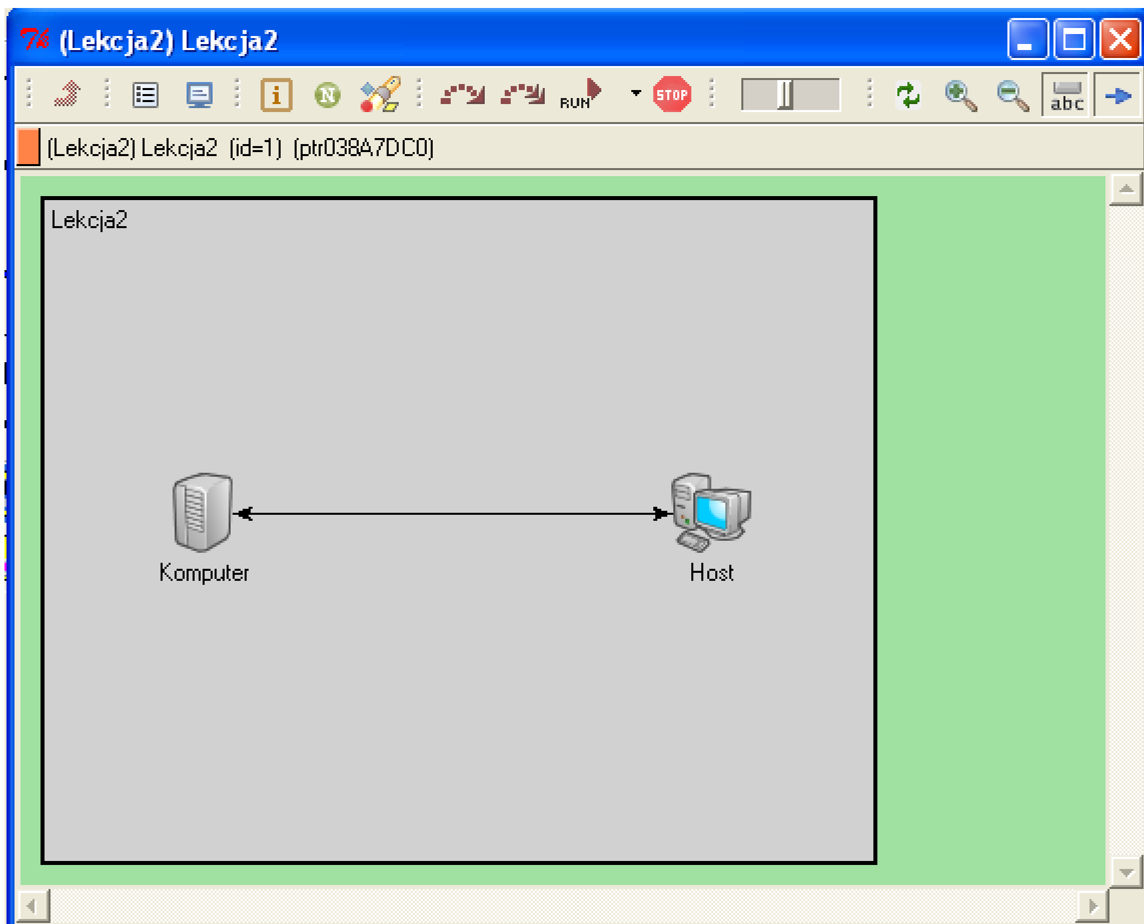
W pliku *omnetpp.ini* należy umieścić sekcje `[Tkenv]`. Najważniejsza opcja tej sekcji to `default-run=1`. Oznacza domyślny wariant przy uruchamianiu symulacji w przypadku, gdy jest więcej niż jeden wariant. Pozostałe opcje są dostępne w oknie głównym symulatora.

Ze względu na możliwość obserwacji przebiegu symulacji uruchomimy nasz przykład w trybie graficznym.

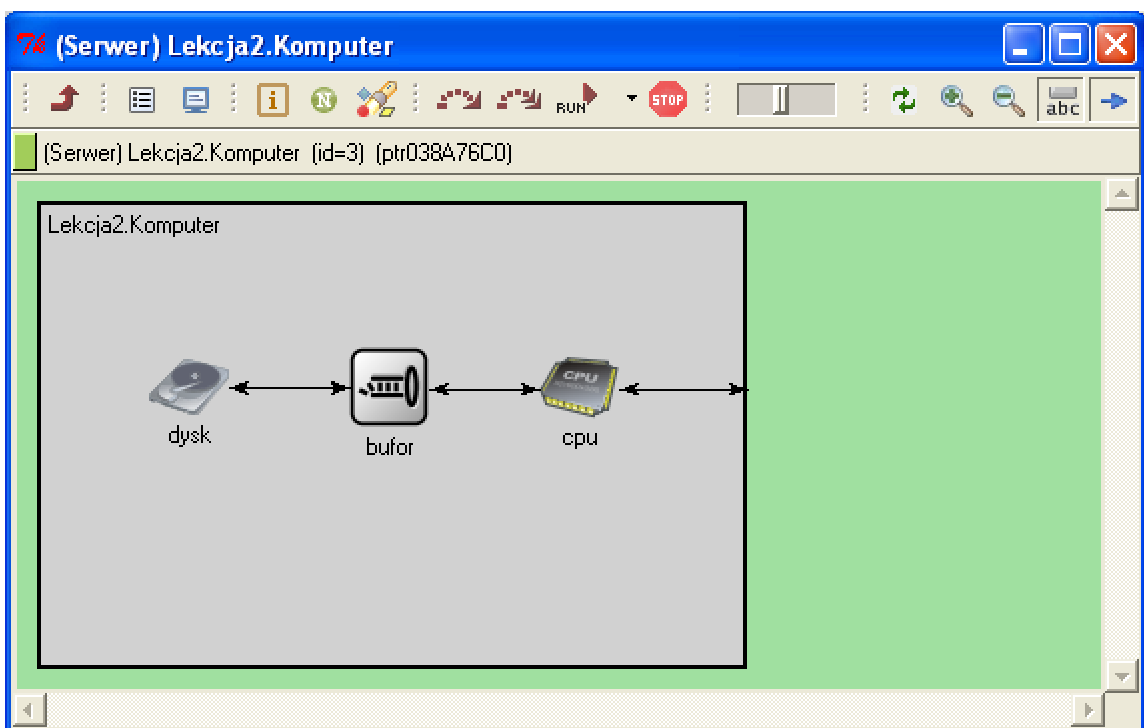
Przy starcie program najpierw czyta plik konfiguracyjny *omnetpp.ini*. Po uruchomieniu tego pliku zobaczymy gotowy program do przeprowadzenia symulacji z trzema okienkami:



Rys. 5. TKEN - Okno główne symulacji

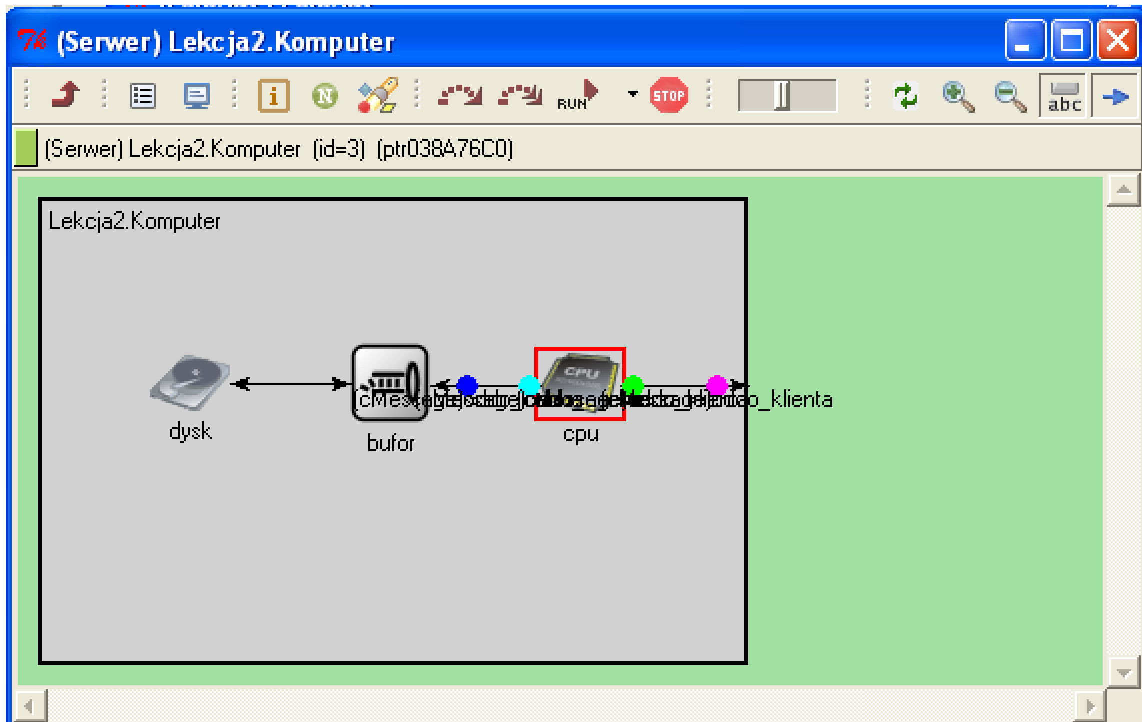


Rys. 6. Okno modułu Symulator



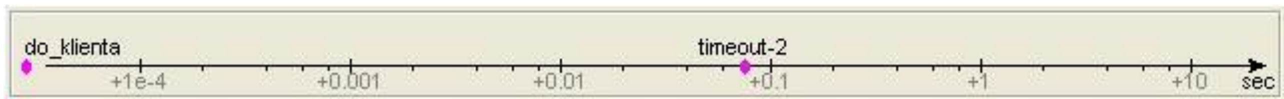
Rys. 7. Okno modułu Komputer

Naciśnij przycisk Run na pasku narzędzi lub F5 w celu uruchomienia symulacji. Powinieneś zobaczyć wysyłanie przez klienta komunikatu do serwera.



Rys. 8. Wysyłanie komunikatów
źródło: wykonanie własne

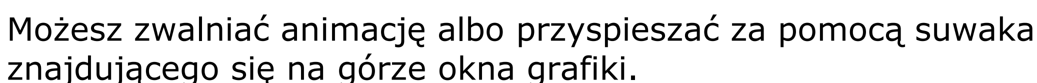
W głównym oknie pokazany jest czas symulacji. Jest to wirtualny czas, który nie ma nic wspólnego z aktualnym czasem (czy zegarem ściennym).



Rys. 8. Linia czasu symulacyjnego
źródło: wykonanie własne

Klikając na punkt na linii czasu wyświetlone zostaną parametry komunikatów, np. nazwa, rodzaj (getKind), długość, czas wysłania oraz czas powrotu komunikatu.

Możesz zwalniać animację albo przyspieszać za pomocą suwaka znajdującego się na górze okna grafiki.



W oknie głównym symulatora masz do wyboru jeden z kilku trybów działania:

Możesz zatrzymać symulację przez wciśnięcie F8 (równoważne przyciskowi STOP na pasku narzędzi), oglądać symulację w pojedynczych krokach (F4 lub STEP), włączyć tryb bez animacji (F6 lub FAST) lub tryb ekspresowy (F7 lub

EXPRES), który zupełnie wyłącza cechy śledzenia dla maksymalnej szybkości. Opcja until powoduje ze symulator działa do ustawionej (w pliku omnetpp.ini) liczby zdarzeń lub czasu symulacyjnego w wybranym trybie normal, fast lub Express.

Na pasku informacyjnym znajduje się (wyliczając od lewego górnego pola) aktualny wariant symulacji i sieć symulacyjna, aktualne zdarzenie w przebiegu symulacji, aktualny czas symulacji oraz obiekt, którego będzie dotyczyć następne zdarzenie, liczba komunikatów na stosie zdarzeń przyszłych, liczba komunikatów utworzonych od początku symulacji, liczba zdarzenia 1 sekundę czasu rzeczywistego, liczbę sekund czasu symulacji w stosunku do czasu rzeczywistego oraz liczbę zdarzeń na 1 sekundę czasu symulacyjnego.

Run #?: Lekcja2	Event #15	T=0.1678439626 (167ms)	Next: Lekcja2.klient_symulatora (id=2)
Msgs scheduled: 2	Msgs created: 12	Msgs present: 5	
Ev/sec: n/a	Simsec/sec: n/a	Ev/simsec: n/a	

Rys. 9. Pasek informacyjny okna głównego Teknv

źródło: wykonanie własne

Możesz wyjść z programu symulacji przez kliknięcie na Close albo wybieranie File|Exit.

Tworzenie własnej klasy dziedziczącej po cPacket

Rozróżnianie źródła wiadomości w modułach może być zrealizowane w bardziej przejrzysty sposób niż zostało to zaprezentowane.

W takim wypadku należy stworzyć plik z rozszerzeniem *.msg zawierający dodatkowe atrybuty przenoszonej wiadomości. Plik taki tworzy się poprzez kliknięcie prawym przyciskiem na Project Explorer w bieżącym projekcie, wybranie nowego pliku Message Definition (msg), któremu następnie należy następnie nadać nazwę (np. `MyPacket.msg`) i wybrać w kreatorze Empty Message File.

W utworzonym pliku dodajemy zmienne adresu źródłowego oraz docelowego.

```
packet MyPacket
{
     int srcAddress;
     int destAddress;
};
```

Po zbudowaniu projektu automatycznie torzone są pliki:

```
MyPacket_m.cc
MyPacket_m.h
```

Z nowej klasy można skorzystać w następujący sposób:

```
#include "MyPacket_m.h"

...
MyPacket *pkt = new MyPacket("pkt");
pkt->setSrcAddress(localAddr);
pkt->setDestAddress(1);
...

int dest = pkt->getDestAddress();
...
```

Adresy źródłowe i docelowe pozwalają na rozpoznawanie komunikatów i przesyłanie ich niezależnie od nadanych im kind'ów.

Sieć z ograniczoną przepustowością łącza

Ograniczenie przepustowości łącza można zrealizować poprzez zdefiniowanie kanału.

```
Channel Ethernet extends ned.DatarateChannel
{
     Datarate = 100Mbps;
}
```

Następnie należy określić, które połączenia w modelowanej sieci mają być ograniczone parametrami zdefiniowanego kanału.

W celu prawidłowego przebiegu transmisji przy zmniejszonej przepustowości łącza należy utworzyć dodatkowy moduł prosty `Siec.ned`, który będzie odpowiedzialny za sprawdzanie czy łącze, przez które ma zostać przesłana wiadomość, jest wolne. W utworzonym module deklarujemy dwie bramy:

```
input wejscie;
output wyjscie;
```

Sprawdzanie zajętości kanału odbywa się w funkcji `activity()` w pliku `siec.cc` w następujący sposób:

```
   cChannel *channel  = gate("wyjscie")->getTransmissionChannel();
   simtime_t txFinishTime = channel->getTransmissionFinishTime();
```

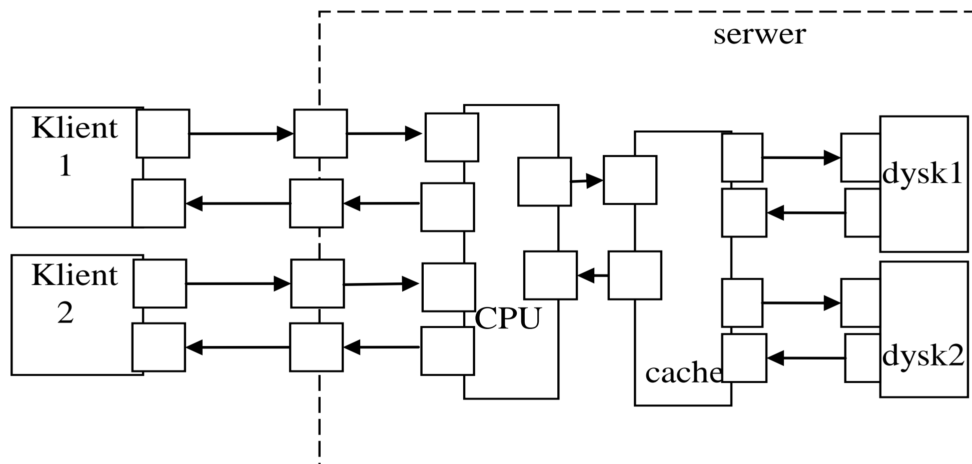
Zmienna `txFinishTime` zawiera informację o symulacyjnym czasie zakończenia bieżącej transmisji. Należy zatem sprawdzić, czy ten czas jest mniejszy od aktualnego czasu symulacji. Jeśli tak jest, to znaczy, że kanał jest wolny i zadanie może zostać wysłane. W przeciwnym wypadku należy odczekać różnicę czasu, a następnie przesłać zadanie. Podczas oczekiwania na zwolnienie kanału, zadania, które przychodzą do modułu, są umieszczane w kolejce.

```
        if (txFinishTime <= simTime())
```

```
    {
         send(zadanie,"wyjscie");
     }
    else
    {
        waitAndEnqueue(txFinishTime-simTime(),&kolejka);
       send(zadanie,"wyjscie");
    }
}
```

Zadanie do samodzielnego przygotowania:

Wykonaj model symulacyjny sieci kolejkowej, w której 2 klientów wysyła żądania do serwera, w skład którego wchodzi procesor, pamięć podręczna cache oraz dwa dyski. Schemat sieci został przedstawiony na rysunku 10.



Rys. 10. Schemat połączeń w sieci

Częstotliwość, z jaką klienci będą wysyłali żądania jest rozkładem wykładniczym o wartości oczekiwanej 0,2s, a wielkość danych, które będą żądali jest rozkładem normalnym z wartością oczekiwaną μ=5000B i wariancją σ2=333B. Proszę nie dopuścić do wysyłania żądań o wielkości mniejszej niż 200B.

Załóżmy, że żądanie trafia w pierwszej kolejności do procesora, a następnie do pamięci cache i ewentualnie dalej do dysku (proszę pamiętać o zaimplementowaniu kolejki w dysku). Procesor obsługując żądania zużywa 527 μs na wyszukanie danych oraz 240 μs na transmisję każdych 512 B danych do klienta.

Prawdopodobieństwo znalezienia danych w pamięci *cache* wynosi 70%. Jeśli żądany plik nie znajduje się w pamięci cache to żądanie idzie do pierwszego dysku z prawdopodobieństwem 30% lub drugiego dysku z prawdopodobieństwem 70%.

Przyjmijmy również, że dysk ma być przeszukiwany w celu znalezienia danych w czasie 5ms, czas transmisji danych przez dysk wynosi 410 μs na każde 4KB

danych.

Sieć, którą będzie połączony klient z serwerem ma przepustowość 100 Mb/s.