Lekcja 5. Funkcje handleMessage() i initialize(), konstruktor i destruktor

W lekcji 5 objaśniane jest działanie i zastosowanie funkcji `handleMessage()` oraz `initialize()`. Omówiony zostanie również konstruktor i destruktor oraz przedstawiony zostanie sposób tworzenia symulacji dającej możliwość wyboru sposobu uruchamiania.

W *lekcji 2* wspomniane było na temat stosowania funkcji `activity()` lub `handleMessage()` do wpływania na przetwarzania zdarzeń zachodzących podczas symulacji. Dla każdego modułu prostego należy zdefiniować jedną z tych funkcji.

W celu zaprezentowania sposobu działania funkcji `handleMessage()` zostanie ona zastosowana w sieci komputerowej z zadania z *Lekcji 4*.
Model symulacyjny zostanie uzupełniony o możliwości wyboru liczby klientów generujących zadania do serwera.

We wcześniejszych lekcjach stosowana była funkcja `activity()`. Ponieważ w tej lekcji zaplanowane jest opracowanie symulacji, pozwalającej na wybór większej liczby klientów, a funkcja `activity()` wymaga definiowania stosów dla każdego modułu, powodowałoby to duże zużycie pamięci. Funkcja `handleMessage()` nie korzysta ze stosu więc będzie odpowiednią do tego typu symulacji.

W funkcji `activity()` można było wpływać na to, kiedy i jak zostanie obsłużona wiadomość natomiast funkcja `handleMessage()` wywoływana jest dla każdej wiadomości przychodzącej do modułu. Poza tym podczas wywołania funkcji `handleMessage()` nie upływa czas symulacyjny dlatego nie można stosować razem z tą funkcja metod `wait()` ani `receive()`.

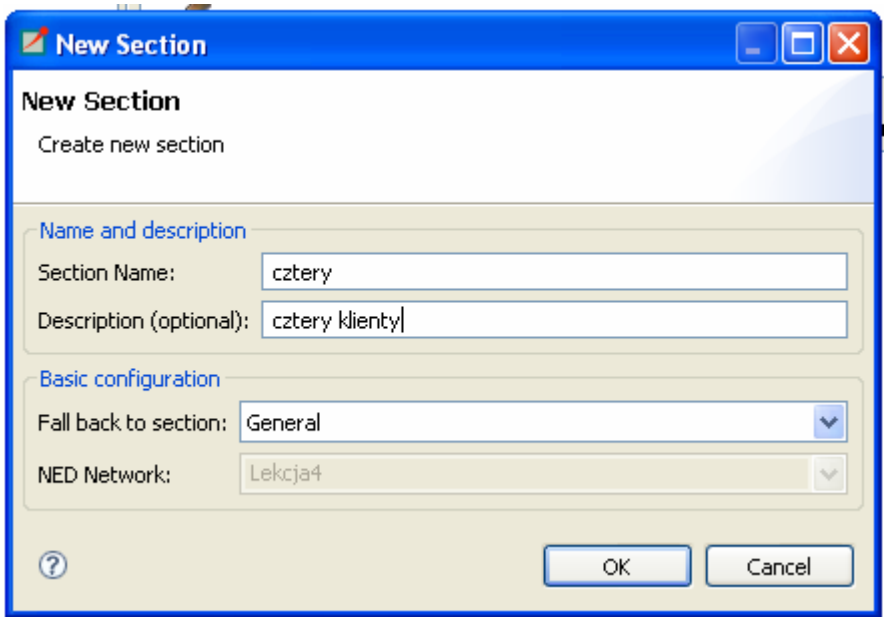
Wraz z funkcja `handleMessage()` można używać metody obsługujące zdarzenia takie jak:

- `send()` – wysyła wiadomości do innych modułów;
- `scheduleAt()` – wysyła wiadomości do siebie;
- `cancelEvent()` – usuwa zdarzenia zaplanowane przy pomocy funkcji `scheduleAt()`;

Ponieważ w zadaniu w module np. `Dysk_twardy` konieczne jest wykorzystanie funkcji `wait()` w celu opóźnień związanych z obsługą wyszukiwania na dysku, a moduły działające na funkcjach

`handleMessage()` i `activity()` mogą być dowolnie mieszane w modelu symulacyjnym, więc w zadaniu zostaną zastosowane obie funkcje.

Zmiany w zadaniu z *lekcji 4* zaczniemy od uzupełnienia pliku *omnetpp.ini* o sekcję `[Config nazwa]` dla poszczególnych uruchomień symulacji. `[Config cztery]` będzie uruchamiał symulacje z 4 klientami, a `[Config pytaj]` będzie pozbawiony wartości `ilosc_klientow`, która odpowiada za liczbę klientów w sieci, co będzie powodowało zapytanie o tą wartość przy starcie programu. W sekcjach tych można umieszczać parametry, których wartość przysłaniają parametry globalne. W wersji 4.0 omneta możemy skorzystać z formularza edycji pliku omnetpp.ini
Wybieramy *Sections*, a następnie *New* i wpisujemy nazwę sekcji i opis (dowolnie).



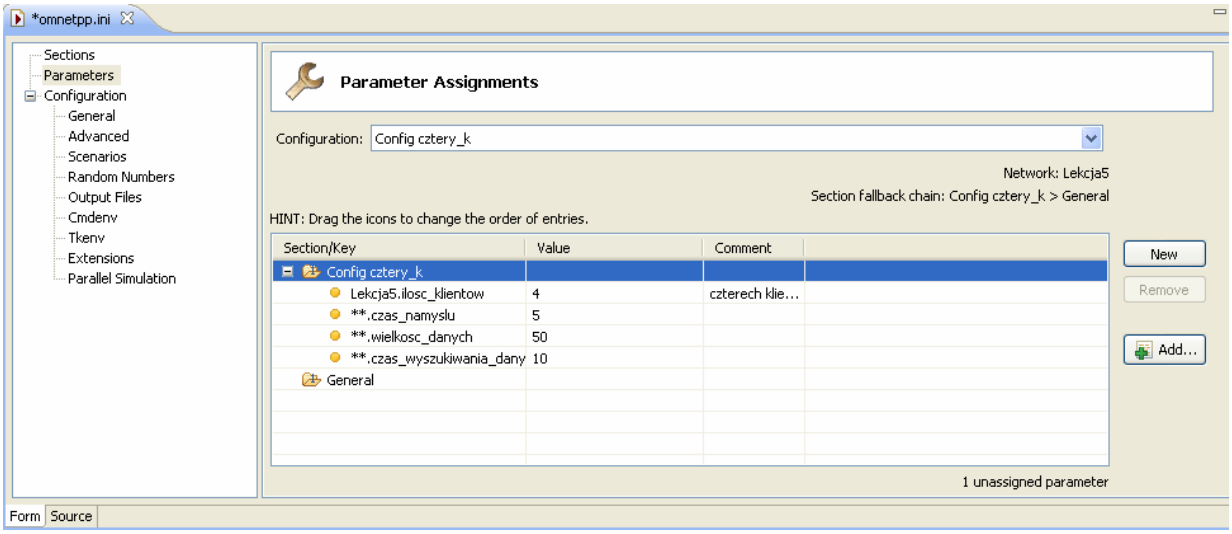
Rys. 1. : Okno dodawania sekcji
Źródło: wykonanie własne

Wybierając source możemy podejrzeć, że omnet samodzielnie wpisał kod do pliku omnetpp.ini.

```
[Config cztery]
description = "cztery klienty"


[Config pytaj]
description = "pyta o liczbe klientow"
```

Następnie przechodzimy do *Parameters,* gdzie możemy przypisać wartości parametrom dla danej konfiguracji. Oprócz liczby klientów podane zostały również inne parametry.



Rys. 2. : Okno dodawania parametrów
Źródło: wykonanie własne

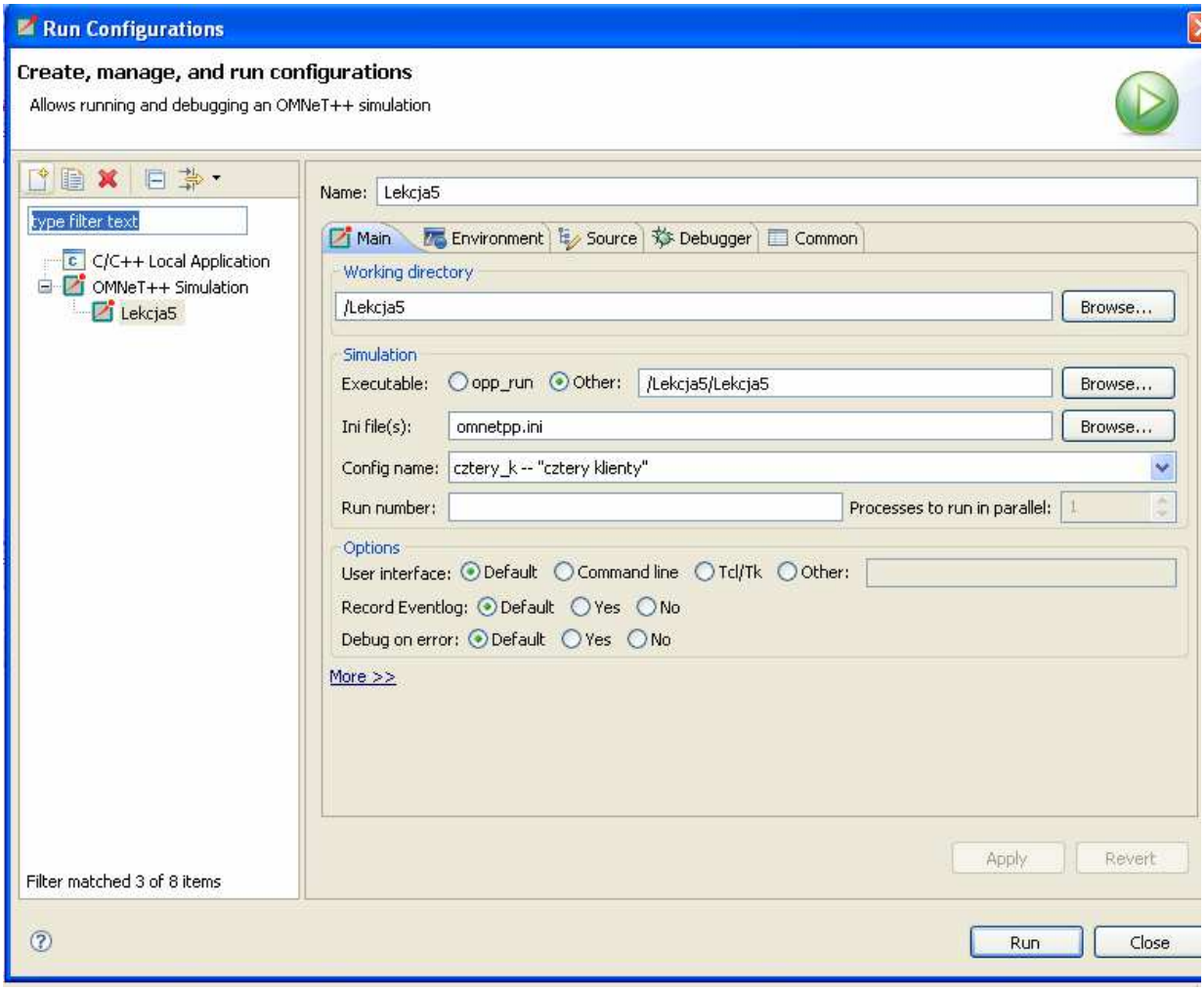
Źródło pliku wygląda teraz następująco:

```
[General]
network = Lekcja5


[Config cztery_k]
description = "cztery klienty"
Lekcja5.ilosc_klientow = 4  # czterech klientow
**.czas_namyslu = 5
**.wielkosc_danych = 50


[Config pytaj]
description = "pyta o liczbe klientow"
```

Prezentowany plik umożliwia uruchamianie konfiguracji zgodnie z potrzebami. Rys. 3 przedstawia okno pozwalające wybrać, którą konfigurację chcemy użyć.



Rys. 3. : Okno uruchamiania konfiguracji
Źródło: wykonanie własne

Ponieważ moduł `Klient` ma za zadanie jedynie wysyłanie komunikatów do switcha nie ma potrzeby używania w nim funkcji `activity()` i definiować dla niego stos. Zastosowana zostanie więc funkcja `handleMessage()` oraz funkcja `initialize()`, w której zostanie zaplanowane wysyłanie wiadomości.
Zastosowany zostanie również konstruktor i destruktor.

```
class Klient : public cSimpleModule
{
  private:
    cMessage *sendMessage;

  public:
    Klient();
    virtual ~Klient();

  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};
```

Funkcja `handleMessage()` wywoływana jest jedynie w przypadku otrzymania wiadomości. Moduł Klient ma za zadanie generować wiadomości wiec konieczne jest zainicjowanie wysyłania wiadomości do samej siebie, czyli `scheduleAt()`, która będzie odpowiedzialna za ciągle planowanie wysłań.

Konstruktor jest wywoływany, kiedy tworzony jest moduł. Ustawia on wartość parametru na NULL. Następnie tuż przed rozpoczęciem wykonania symulacji wywołana jest funkcja `initialize()`.

Destruktor zastosowany jest w celu usunięcia wszystkiego, co zostało zaalokowane funkcją *new* i co znajduje się w module klasy. Do usunięcia własnych wiadomości używa się funkcji `cancelAndDelete(msg)`.

```
Klient::Klient()
{
    sendMessage = NULL;
}

Klient::~Klient()
{
    cancelAndDelete(sendMessage);
}
```

Wewnątrz funkcji `initialize()` zaplanowane jest pierwsze wysłanie wiadomości powodujące pierwsze wywołanie funkcji `handleMessage()`.

```
void Klient::initialize()
{
    sendMessage = new MyPacket ("sendMessage");
    scheduleAt(0.0, sendMessage);
}
```

Następnie konieczne jest zaplanowanie przyszłych zdarzeń dla modułu w funkcji `handleMessage()`. Używa się w tym celu funkcji

scheduleAt(simTime()+delta), która jako parametr akceptuje całkowity czas symulacji.

Implementując funkcje handleMessage() należy pamiętać o sprawdzeniu czy przychodząca wiadomość jest własną wiadomością czy przysłaną przez inny moduł.
W przypadku modułu Klient komunikaty przybyłe od modułu Switch nie są obsługiwane tylko usuwane. Jeśli komunikat jest wiadomością zaplanowaną przez moduł Klient to sprawdzany jest identyfikator bramy połączonej z bramą wyjściową Klienta, i ta wartość ustawiana jest jako kind, a następnie wysłana w wiadomości do switcha.

```
void Klient::handleMessage(cMessage *msg)
{
     char nazwa[20];
     int wielkosc_danych = par ("wielkosc_danych");

     strcpy(nazwa,"od_klienta");

     MyPacket *zadanie = new MyPacket (nazwa);
     int id_klienta = gate("od_klienta")->getOwnerModule()->
getindex()+1;

     //spr czy wiadomość jest wiadomością wyslana do siebie
     if (msg==sendMessage)
     {
        zadanie->setKind(id_klienta);
        zadanie->setSrcAddress(id_klienta);
        zadanie->setDestAddress(1);
        zadanie->setByteLength(wielkosc_danych);
        ev << "Klient do switcha zadanie z kind=" <<
           zadanie->getKind() << " ,Wielkosc danych= " <<
           wielkosc_danych << endl;

        send(zadanie, "od_klienta");
        scheduleAt(simTime()+(double)par("czas_namyslu"),
           sendMessage);
     }
     else
          delete msg;
}
```

Możliwe jest również zastosowanie funkcji handleMessage() w module Sterowanie_switchem, którego zadanie polega jedynie na zmianie adresu docelowego i odesłanie wiadomości z powrotem do Magazynu.

```
class sterowanie_switchem: public cSimpleModule
{
public:
```

```
    MyPacket *zadanie;
protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};
```

Ponieważ moduł ma reagować tylko na wiadomości pochodzące od Magazynu, planowanie zdarzeń nie jest potrzebne.

```
void Sterowanie_switchem::handleMessage( cMessage *zadanie)
{

   zadanie = (MyPacket *) msg;
   ev << "Sterowanie dostalo od Magazynu zadanie z
kind="<<zadanie->getKind()<< endl;

   int dest = zadanie->getDestAddress();
   zadanie->setDestAddress(dest+1);
   send (zadanie, "do_portow");
   ev << "Sterowanie do Magazyn zadanie z kind=" << zadanie->
getKind() << endl;
}
```

Sposób wykorzystania funkcji handleMessage() zostanie również pokazany w module Procesor. Działanie funkcji handleMessage() w tym module jest odmienne ze względu na potrzebę poprawnej obsługi kolejki.

```
class Procesor: public cSimpleModule
{
     protected:
          virtual void handleMessage(cMessage *msg);
     public:
        MyPacket *zadanie;
        cQueue kolejka;
        virtual void wysylamy_na_dysk(cMessage MyPacket *);
        virtual void wysyla_do_switcha_od_serwera(cMessage
MyPacket *);
};
```

Podobnie jak w przypadku modułu Sterowanie_switchem, moduł Procesor reaguje jedynie na komunikaty przybyłe od innych modułów więc nie ma potrzeby definiowania funkcji initialize(), ani planowania zdarzeń za pomocą scheduleAt().
Na początek należy sprawdzić czy wiadomość jest wiadomością własną (isSelfMessage) lub też przybyła od dysku albo od switcha. Następnie sprawdzane jest czy kolejka jest pusta, jeśli tak to wykonywane jest działanie na przybyłej wiadomości. Natomiast w przypadku istnienia zadań w kolejce, przybyłe zadanie wrzucane jest na koniec kolejki za pomocą

nazwa_kolejki.insert(msg), a obsłużone jest zadanie pobrane z kolejki za pomocą metody pop().

```
void Procesor::handleMessage(cMessage *msg)
{
    kolejka.setName("kolejka");

    if (msg->isSelfMessage())
    {
        char nazwa[20];
        strcpy(nazwa, msg->getName());

        zadanie = (MyPacket *) msg;

        if(strcmp(nazwa,"do_dysku")==0)
        {
            ev << "Procesor do dysku zadanie z kind=" << msg-
>getKind() << endl;
            send (zadanie, "do_dysku");
        }
        else if(strcmp(nazwa,"do_switcha")==0)
        {
            ev << "Procesor do switcha zadanie z kind=" <<
msg->getKind() << endl;
            send(zadanie,"do_klienta");
        }

       //sprawdzenie czy jest cos w kolejce

        if (!kolejka.empty())
               {
                   zadanie = (MyPacket *)kolejka.pop();

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

                   //spr czy od switcha
                   if (dest == 4)
                   {
                       ev << "Procesor dostal od switcha
zadanie z kind=" << zadanie->getKind() << endl;
                       wysylamy_na_dysk(zadanie);
                   }

                  //spr czy z dysku
                   else
                   {
                       ev << "Procesor dostal od dysku zadanie
z kind=" << zadanie->getKind() << endl;
                       wysyla_do_switcha_od_serwera(zadanie);
                   }
```

```
             }

    }
    else
    {
        int dest;
            // jesli kolejka jest pusta to odbiera zadanie za
pomoca funkcji receive
             if (kolejka.empty())
                 zadanie=(MyPacket *) msg;

             else // w przeciwnym wypadku pobiera zadanie z
konca kolejki
             {
                 zadanie = (MyPacket *)kolejka.pop();
                 kolejka.insert(msg);
             }

             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);
             }
    }
}
```

Jeśli w ramach jednego moduły wysyłanych jest kilka wiadomości własnych, to warto nadać im odpowiednie nazwy, aby w łatwy sposób rozpoznać czego dotyczą, np. w metodzie wysyłającej na dysk można to rozwiązać następująco:

```
void Procesor::wysylamy_na_dysk(MyPacket *zadanie)
{
    char nazwa[20];
    strcpy(nazwa, "do_dysku");

    double czas_wyszukiwania_danych =
(double)par("czas_wyszukiwania_danych");
    sendMessage = zadanie;
    sendMessage->setName(nazwa);
```

```
    scheduleAt(simTime()+czas_wyszukiwania_danych,
sendMessage);
}
```

Zadanie do samodzielnego przygotowania:

W programie opracowanym samodzielnie w ramach zadania 4 zmienić wszystkie moduły, poprzez zamianę funkcji `activity` na `handleMessage`, tak, by nie zmienić funkcjonalności programu.