СУЧАСНІ ПРОГРАМНІ ЗАСОБИ ОПТИМІЗАЦІЇ ТА МОДЕЛЮВАННЯ ИНФОКОМУНІКАЦІЙНИХ МЕРЕЖ

ЗМІСТ

ВСТУП

Традиційні технології аналізу, оцінки ефективності функціонування ТКС та вирішення задач управління їх ресурсами, засновані на використанні математичних моделей систем масового обслуговування (СМО) [1-3]. При цьому найбільш істотні результати теорії масового обслуговування (ТМО) пов'язані з використанням марківських моделей випадкового процесу [4-8], що описує вхідний потік, а також процеси обслуговування та очікування. Ці методи можуть бути успішно застосовані в припущенні про пуассонівський характер вхідного потоку.

Наявна схожість процесів функціонування мереж та систем масового обслуговування визначає можливість використання для аналізу мереж добре розвиненим математичним апаратом теорії масового обслуговування [185, 186]. Ця теорія спирається на потужний математичний апарат марківських випадкових процесів [107, 187]. При цьому фактично, використовується допущення про те, що розподіл інтервалів між запитами та тривалістю їх обслуговування є експонентними. Відповідні математичні моделі обслуговування систем виявляються простими та елегантними. Крім того, використання припущення про посилення вхідного потоку запитів, а також потоку замовлених запитів задає нижні, песимістичні оцінки ефективності систем обслуговування (з усіх випадкових потоків з заданою інтенсивністю пуассоновий потік є найбільш випадковим.) Ці обставини визначають і пояснюють широке застосування марківських моделей систем обслуговування. З використанням цих моделей отримані прості аналітичні співвідношення для розрахунку розподілів можливостей станів багатоканальних систем обслуговування з відхиленнями, з очікуванням при обмеженні по довжині черги та тривалості очікування в черзі, а також для систем без втрат.

Засобами стандартних марківських моделей ТМО вирішуються всі завдання аналізу одно- і багатоканальних систем, на вхід яких надходить однорідний потік запитів або суперпозиція потоків без пріоритетів [1]. Однак, у завданнях з приорітетами повні результати відомі тільки в тому випадку, коли сумарний неоднорідний потік надходить в одноканальну систему. Складність вирішення цієї задачі для багатоканальної системи з неоднорідним потоком визначається швидким зростанням її розмірів. Так для звичайних ординарних потоків на вході та кількості каналів системи розмір завдання досягає десятків або сотень тисяч. Тому єдиний ефективний шлях вирішення таких завдань

полягає у використанні методу фазового збільшення стану, суть якого полягає в декомпозиційному підході.

При оцінці продуктивності мережної інфраструктури, в першу чергу, слід розрізняти реальну завантаженість мережі, визначену за даними моніторингу трафіку, і пікову завантаженість. Під піковою завантаженістю слід розуміти завантаженість, що виникає в ситуації, коли всі робочі станції, використовуючи будь-які доступні види трафіку, обмінюються даними з усіма комп'ютерами та серверним устаткуванням. Якщо при розробці проекту модернізації мережі на основі даних, отриманих в результаті моделювання, орієнтуватися на пікові навантаження, то обсяг необхідних матеріальних засобів для модернізації буде надмірно великим. З упевненістю можна стверджувати, що така ситуація практично ніколи не з'явиться, але, змоделювавши її, можна виявити потенційно вузькі місця при розробці проекту мережної інфраструктури.

Найбільш відповідною є модель реальної завантаженості мережі. У цій моделі можна змінювати багато параметрів мережного обміну даними, що призводить до появи декількох варіантів вирішення заданого завдання. Така модель дозволяє моделювати роботу мережі в динаміці, тобто оцінювати її ефективність залежно від поточного часу або доби. Це допоможе створити більш ефективні рішення щодо оптимізації архітектури мережі.

1. ОГЛЯД СИСТЕМ ІМІТАЦІЙНОГО МОДЕЛЮВАННЯ ІНФОКОМУНІКАЦІЙНИХ МЕРЕЖ

Основним інструментом аналізу інфокомунікаційних мереж є імітаційне моделювання в комп'ютерних програмах-симуляторах без застосування реального обладнання. На етапі побудові моделі виникає питання про вибір інструментарію. При цьому важливими потребами для створення ефективної моделі є:

— детальна реалізація протоколів мереж;

— можливість написання і підключення власних модулів;

— можливість зміни параметрів моделювання під час проведення експериментів;

— платформна незалежність;

— розвинений графічний інтерфейс;

— ціна продукту.

Найбільш популярними середовищами імітаційного моделювання є:

- The Network Simulator (NS-2) - об'єктно-орієнтований програмний продукт, ядро якого реалізовано мовою C ++. На базі NS-2 можлива організація наочної демонстрації функціонування протоколів і мережевих механізмів.
- OPNET Modeler Suite (OPNET) - засіб для проектування і моделювання локальних і глобальних мереж, комп'ютерних систем, додатків і розподілених систем. Включає наступні продукти: Netbiz (проектування і оптимізація обчислювальної системи), Modeler (моделювання і аналіз продуктивності мереж, комп'ютерних систем, додатків і розподілених систем), ITGuru (оцінка продуктивності інфокомунікаційних мереж і розподілених систем).
- • OMNeT++ здатний до розширювання, модульний фреймворк, на основі компонентів та бібліотек на мові C++, який використовується для побудови моделей мереж, та являє собою симулятор дискретних подій.

- До системи OMNeT++ закладено детальну реалізацію 24-х протоколів передачі різних рівнів моделі ISO-OSI, при чому є можливість написання і підключення власних додаткових модулів та підмодулів. У системі присутній розвинений графічний інтерфейс.

Надалі буде дано більш детальний огляд систем моделювання інфокомунікаційних мереж.

1.1 Програмний комплекс Network Simulator (NS-2)

Об'єктно-орієнтований програмний продукт Network Simulator (NS-2) має ядро, яке реалізовано на мові C ++. У якості інтерпретатора використовується мова скриптів (сценаріїв) OTcl (Object oriented Tool Command Language). Комплекс NS-2 повністю підтримує ієрархію класів мови C++ (у термінах NS-2 зветься компільованою ієрархією) та ієрархію класів інтерпретатора OTcl (у термінах NS-2 зветься інтерпретованою ієрархією).

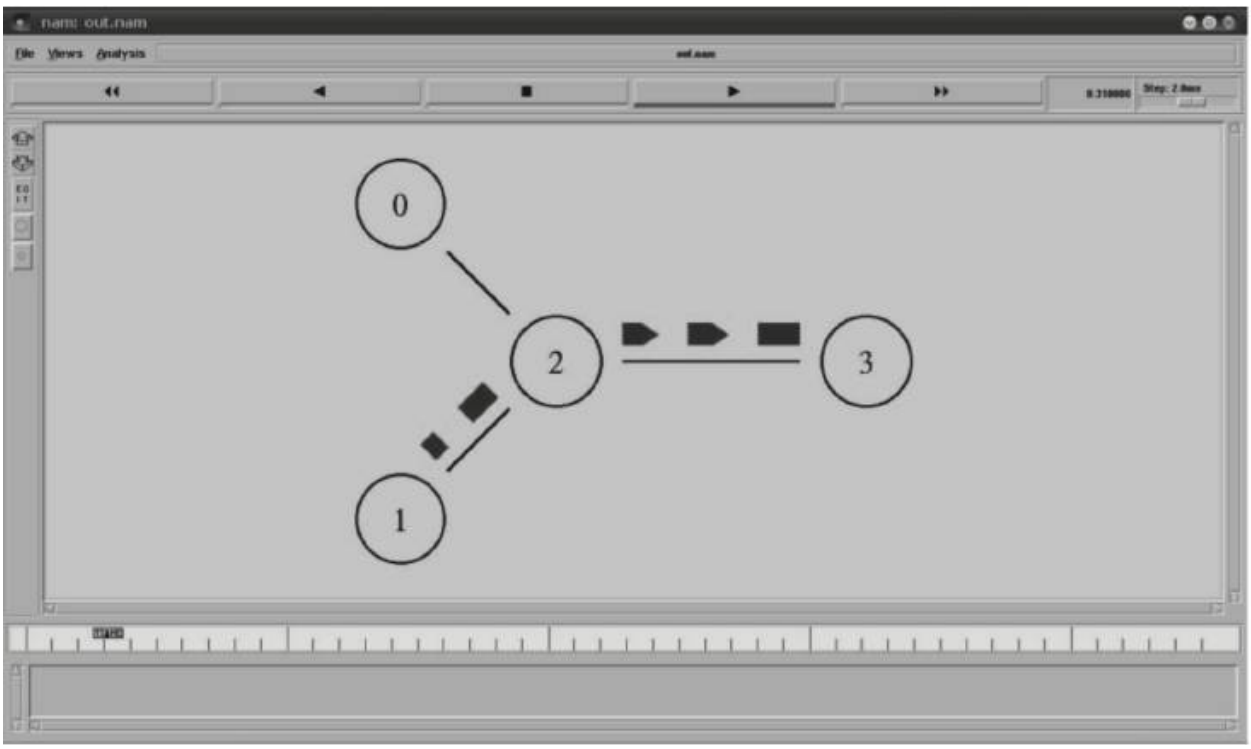

Рис. 1.1. Екрана форма головного вікна додатку NS-2

Використання двох мов програмування у додатку NS-2 пояснюється наступними причинами. З одного боку, для детального моделювання протоколів необхідно використовувати системну мову програмування, що забезпечує високу швидкість виконання і здатність маніпулювати досить великими обсягами даних. З іншого боку, для зручності користувача і швидкості реалізації і модифікації різних сценаріїв моделювання привабливіше використовувати мову програмування більш високого рівня абстракції. Такий підхід являє собою компроміс між зручністю використання і швидкістю. В NS-2 у якості системної мови використовується C++, що забезпечує наступне:

— високу продуктивність;

— роботу з пакетам потоку на низькому рівні абстракції моделі;

— модифікацію ядра NS-2 з метою підтримки нових функцій і протоколів.

У якості мови програмування високого рівня абстракції використовується мова скриптів OTcl, яка дозволяє забезпечити ряд позитивних властивостей, які надає мова Tcl/Tk (OTcl є об'єктно-орієнтованим розширенням мови Tcl):

— простоту синтаксису;

— простоту побудови сценарію моделювання;

— можливість з'єднання воєдино блоків, виконаних на системних мовах програмування і просту маніпуляцію ними.

Об'єднання з метою спільного функціонування C++ та OTcl виконуєтья за допомогою TclCl (Classes Tcl). TclCl являє собою інтерфейс між об'єктами мови C++ і мови OTcl, яким користуються NS-2. Однак, архітектура NS-2 не оптимальна і вимагає доопрацювань. Зокрема, поєднання у колективу модель об'єктів C++/Otcl створює деякі ускладнення: По-перше, мала поширеність і документування мови OTcl, а також складність налагодження написаних з її допомогою скриптів. По-друге, існують обмеження на спільне використання об'єктів C++. По-третє, труднощі із ефективністю використання пам'яті, симулятор повинен підтримувати обидва потоки даних - повний і обмежений.

На даний момент симулятор NS-2 оптимізований для обмежених потоків даних, трасування, спрощення роботи з моделями з великою кількістю елементів і зменшення обсягу реєстрованих подій, які тросуються,.

Для отримання статистики моделювання додаток повинен гнучко конфігуруватися користувачем. Деякі можливості вже присутні в симуляторі NS-2: можливість трасування окремих пакетів або стеження за певним каналом. Більш того, необхідно дозволити користувачеві визначати тип реєстрованих даних, наприклад, чи потрібно реєструвати порти джерела і приймача при дослідженні протоколу TCP.

1.2 Програмний комплекс OPNET

OPNET Modeler пропонує користувачам графічне середовище для створення, виконання та аналізу моделювання подій у інфокомунікаційних мережах. Програмне забезпечення може бути використано для великого ряду завдань, наприклад, типове створення і перевірка протоколів зв'язку, аналіз взаємодій протоколів, оптимізація та планування мереж. Також є можливість здійснити за допомогою пакетів перевірку правильності аналітичних моделей, і опис протоколів. Загальний вигляд головного вікна системи показано на рис. 1.2.

В рамках, так званого, редактора проекту можуть бути створені різні за складністю мережеві об'єкти, у яких користувачем може бути використано та поєднано різні форми комутації вузлів і зв'язків. У додатку реалізовано автоматизовану генерацію мережевої топології - кільце, зірку, випадкову мережу. Підтримується і реалізується утилітами додатку імпортування мережевих топологій в різних форматах. У OPNET Modeler може бути автоматично згенеровано випадковий трафік за алгоритмами, які задано користувачем, чи за алгоритмами, які входять до стандартної комплектації додатка, які імітують реальний трафік мережі.

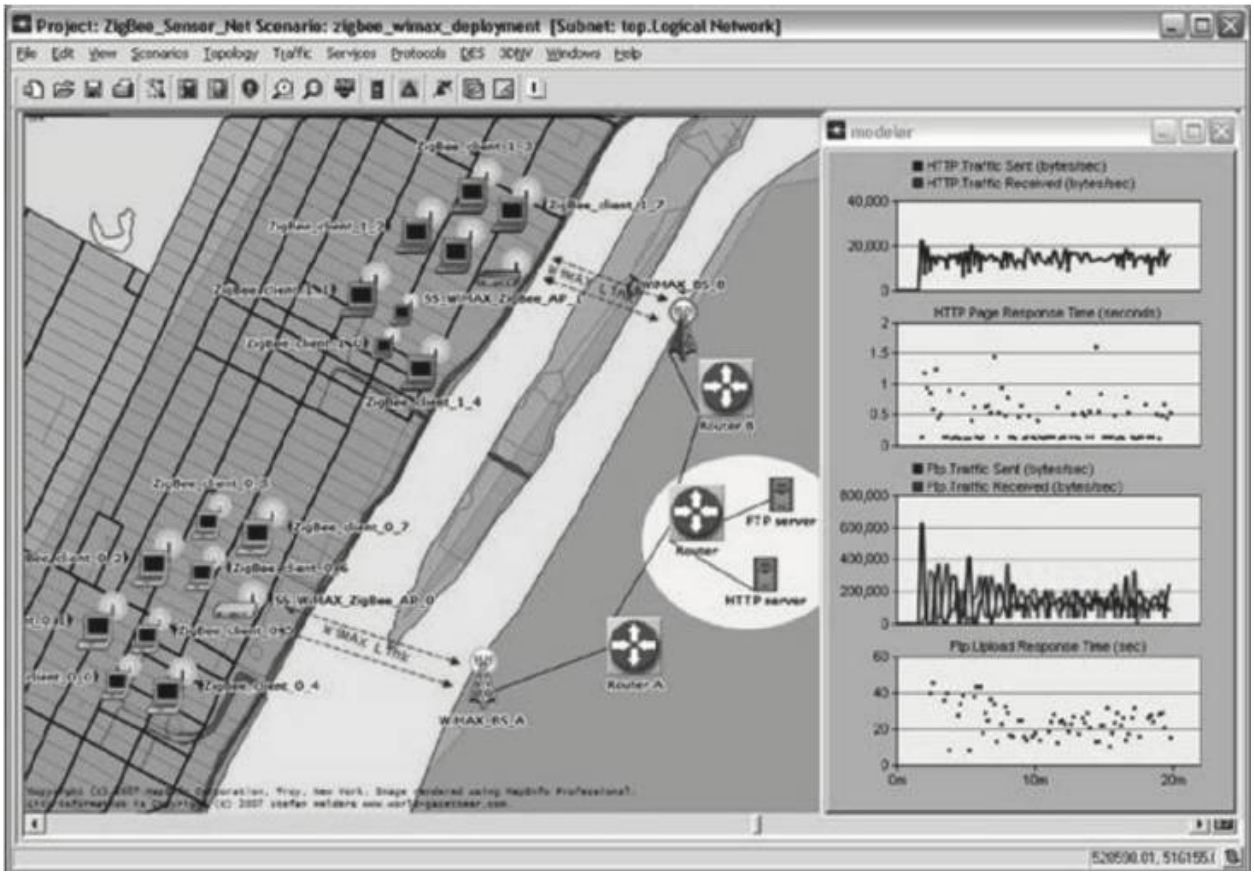

Рис. 1.2. Екранна форма головного вікна OPNET Modeler

Процес моделювання може бути проаналізований користувачем – додаток генерує графи і анімацію трафіка автоматично.

Нова особливість додатку - це автоматичне перетворення звітів у формат html. Одним з плюсів створення моделі мережі за допомогою програмного забезпечення OPNET є те, що рівень гнучкості, який забезпечується ядром моделювання, є таким же, як і для моделювання, написаних з нуля моделей, але об'єктна побудова середовища дозволяє користувачеві набагато швидше робити розробку, удосконалення та виробляти моделі для багаторазового використання. Є декілька різних середовищ у стандартному редакторі - по одному для кожного типу об'єктів. Організація цих об'єктів має ієрархічну структуру. Мережні об'єкти (моделі) пов'язують вузли мережі об'єктами комутації, при цьому, об'єкти вузлів складаються із різних типів модулів: черг пакетів, модулів процесорів, передавачів і приймачів.

Програмне забезпечення для моделювання радіоканалу містить моделі антени радіопередавача, антени приймача, об'єкти вузлів, яки можуть переміщуватись, включаючи супутники. Логіку поведінки процесора і модулів черг пакетів визначає модель процесу, яку користувач може створювати і змінювати в межах редактора процесу, в якому, у свою чергу, користувач може визначити модель процесу через комбінацію алгоритму роботи кінцевого автомата (finite-state machine - FSM) і операторів мови програмування C/C ++.

Виклики подій моделі процесу протягом моделювання керуються за рахунок порушення переривання, а кожне переривання, у свою чергу, відповідає події, яка повинна бути обробленою моделлю процесу.

Основу зв'язку між процесами представляє собою структура даних, яка називається пакетом. У додатку можуть бути задані різні формати пакета. Вони визначають, які поля можуть містити такі стандартні типи даних, як цілі числа, числа з плаваючою комою і покажчики на пакети (ця здатність дозволяють інкапсулювати моделювання пакета). Структура даних, яка викликає інформацію з контролю за інтерфейсом (interface control information - ICI), може бути розділена між двома подіями моделей процесу - це ще один механізм для міжпроцесорного зв'язку, що зручно для команд моделювання і відповідає архітектурі багаторівневого протоколу.

Кожен процес, також, може динамічно породжувати дочірні процеси, що спрощує функціональний опис таких систем, як сервери. Кілька основних моделей процесу входять до базової комплектації пакета, це дозволяє моделювати популярні протоколи роботи з мережами та алгоритми, на приклад:

— протокол шлюзу кордону (border gateway protocol - BGP);

— протоколу контролю передачі;

— Інтернет протокол (TCP/IP);

— ретрансляції кадрів (frame relay);

— протокол Ethernet;

— протокол асинхронного режиму передачі (asynchronous transfer mode - ATM);

— алгоритм WFQ (weighted fair queuing).

Базові моделі корисні для швидкого розвитку складних імітаційних моделей для загальних архітектур інфокомунікаційни мереж, а також для навчання студентів, щоб дати точний функціональний опис протоколів. Існує можливість супроводу розробки коментарями і графікою (з підтримкою гіпертексту) моделей мереж, вузла або процесу. Одним з недоліків системи є велика вартість на представлений продукт.

1.3 Програмний комплекс OMNeT++

Система OMNeT++ являє собою симулятор дискретних подій. Зміна стану модельованої системи відбувається в дискретні моменти часу відповідно до списка майбутніх подій (future eventlist), які відсортовано за часом. Подією може бути: початок передачі пакета, таймаут і т.п. Події відбуваються на основі виконання простих модулів (simple module). У такого модуля є функції ініціалізації, обробки повідомлення, дії і завершення роботи.

Система INET Framework - це комплект модулів з відкритим вихідним кодом, які дозволяють реалістично моделювати вузли мереж та протоколи дротових і бездротових мереж. Він включає моделі різних протоколів Інтернету: IP, IPv6, TCP, UDP, 802.11, Ethernet, PPP, MPLS з LDP і RSVP-TE signalling, OSPF і ряд інших. У комплект також входять різні реалістичні приклади використання цих протоколів.

Найвищий рівень абстракції в моделюванні IP в INET Framework предсталено мережею, яка складається з IP-вузлів. Вузол може бути маршрутизатором або хостом. IP-вузол відповідає комп'ютерному поданню стека протоколів Інтернет. Модулі, з яких він складається, організовані таким чином, щоб моделювати обробку IP-дейтаграм в операційних системах.

Обов'язковим є модуль, який відповідає за мережевий рівень (який реалізує обробку IP) і модуль "мережевий інтерфейс". Додатково підключаються модулі, що реалізують протоколи транспортного рівня.

В основному вікні візуалізації (рис. 1.3) показано комп'ютерну мережу для якої проводять моделювання. Вона являє собою набір клієнтських станцій, сервер і маршрутизатор, з'єднаних каналами зв'язку. Кожен вузол може нести різну функціональність в залежності від параметрів або набору внутрішніх модулів. Внутрішні модулі відповідають за роботу протоколів і додатків на різних рівнях моделі OSI.

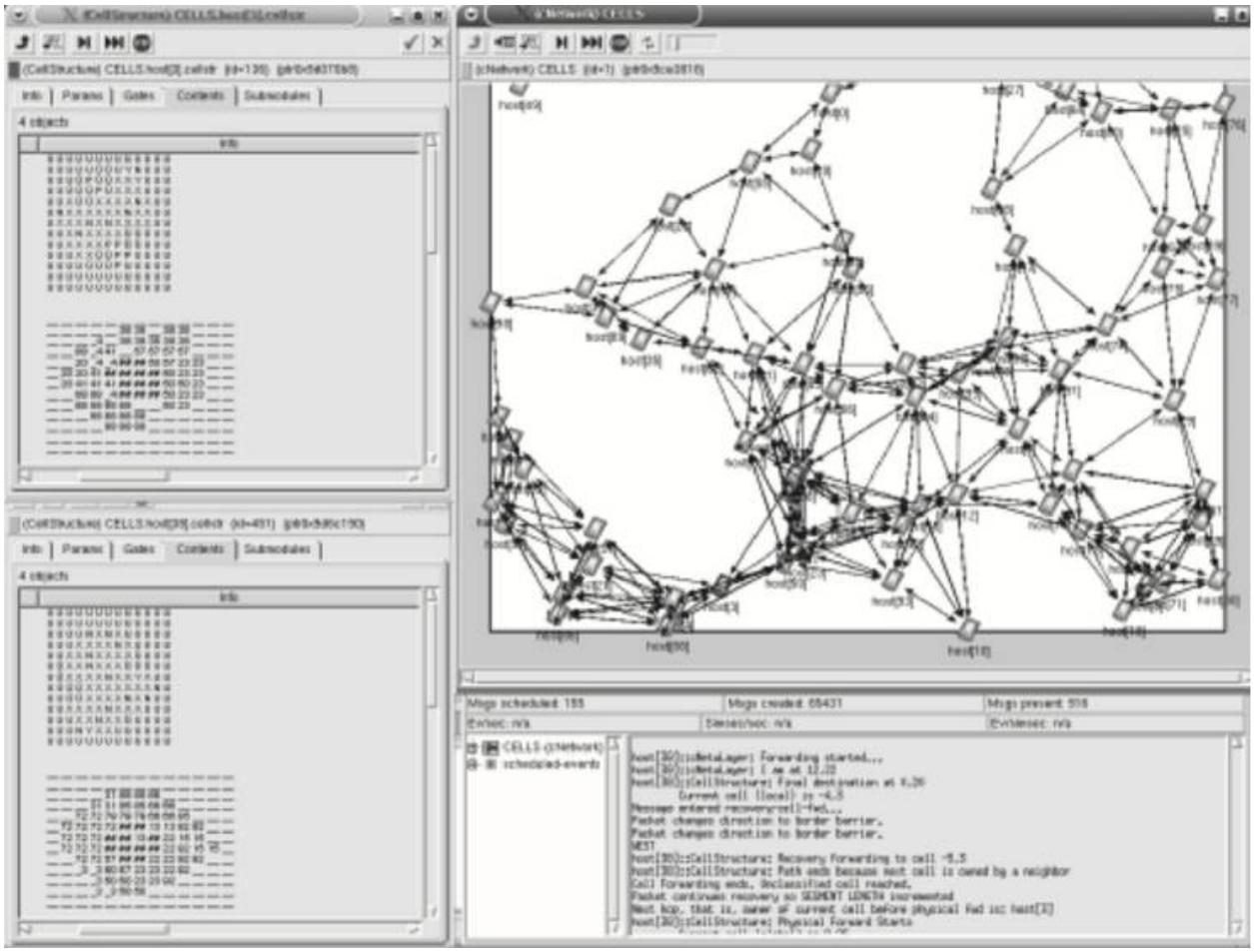

Рис. 1.3. Екранна форма головного вікна OMNeT++

Вузли мережі з'єднуються між собою каналами зв'язку, параметри котрих можна змінювати. На основному вікні візуалізації (рис. 1.3) відображається

внутрішня структура сервера і маршрутизатора (протоколи мережевого, транспортного рівнів моделі OSI, таблиця маршрутизації і вказано запис логу всіх процесів, які відбуваються. Всі вищезгадані властивості доступні в будь-який час протікання процесу. Тим самим, користувач може відстежити пересування пакета, який його цікавить, по всій мережі.

1.4. Програмний комплекс javaNetSim

Основним призначенням симулятора javaNetSim є імітація роботи всіх рівнів стека протоколів TCP/ IP. Для цього в ньому імітується робота протоколів кожного з рівнів, чим досягається повна імітація роботи мережі. Таким чином симулятор javaNetSim є зручним для виконання лабораторних робіт студентів.

Симулятор javaNetSim є об'єктно-орієнтованим, він написаний мовою Java, є кросплатформним, тобто javaNetSim працює під будь-якомою операційною системою, де є віртуальна Java-машина.

Архітектура симулятора javaNetSim має наступний вигляд. В його основі лежить клас Simulation (Імітація), який містить об'єкти класів Link (Лінія зв’язку) і Node (Вузол мережі). Цей клас призначений для об'єднання пристроїв і ліній зв'язку в єдину мережу. Клас Link містить канали зв’язку для об'єктів класу Node, і призначений для з'єднання двох вузлів між собою.

Клас Node містить зв’язки з об'єктами класу Link і є найбільш загальною моделлю мережевого пристрою.

Всі реальні мережеві пристрої є похідними від об'єкта класу Node і відповідають моделі стека протоколів TCP/IP:

— Hub (Концентратор) - DataLink Layer Device (Пристрій фізичного рівня) - має п'ять портів, тобто до нього можливо підключити до п'яти ліній зв'язку;

— Router (Маршрутизатор) - Network Layer Device (Пристрій мережевого рівня) - має два порти, а також стек протоколів TCP / IP (ProtocolStack);

— PC (ПК) - Applications Layer Device (Пристрій рівня додатків) - має один порт, стек протоколів TCP/IP, а також можливість виконувати клієнтську або серверну частину будь-якої програми.

Для взаємодії з користувачем та візуалізації кожному мережевому пристрою потрібний графічний відповідник. Його забезпечують наступні класи:

— GuiHub (Графічний інтерфейс концентратора);

— GuiRouter (Графічний інтерфейс маршрутизатора);

Як самі мережеві пристрої, так і графічний користувальницький інтерфейс мережевих пристроїв повинен бути єдиним. Цим об'єднанням займається клас SandBox (Робоча область).

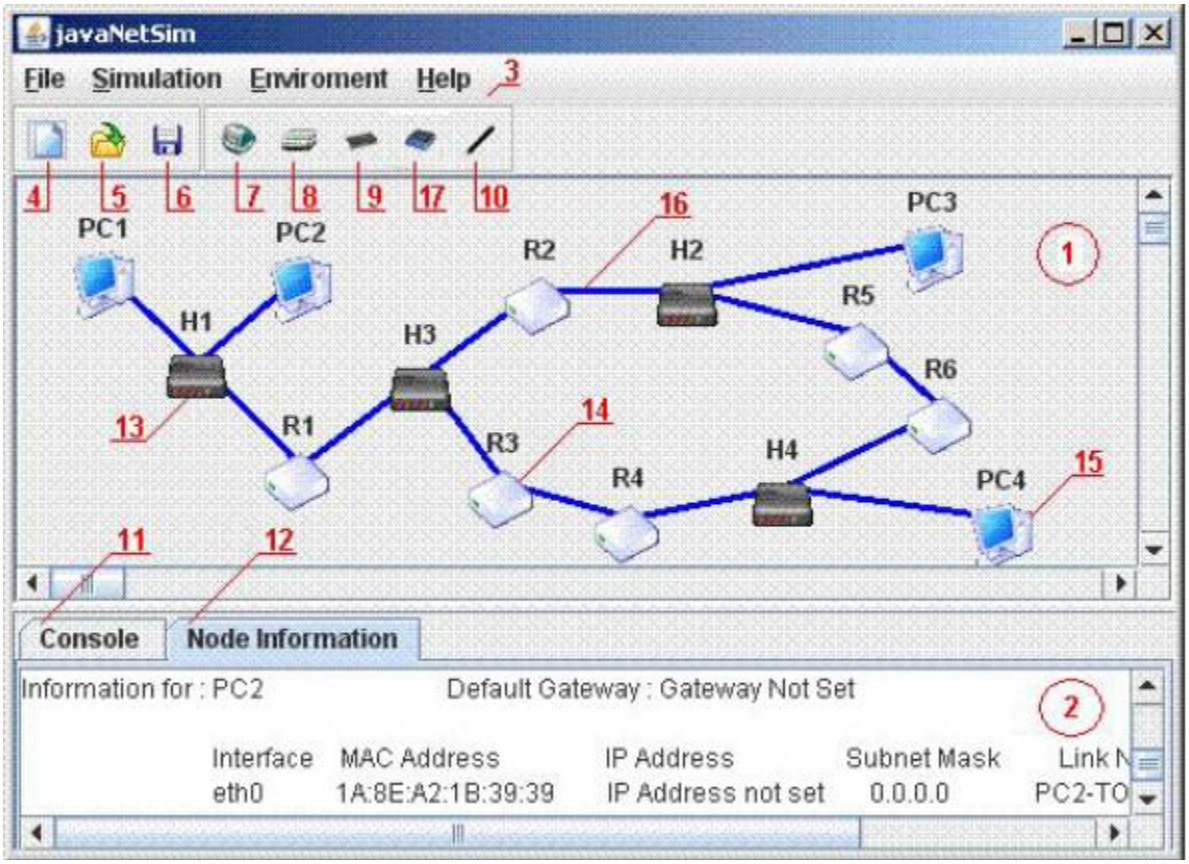

Рис. 1.4. Екранна форма головного вікна javaNetSim

1.5. Програмний комплекс COMNETIII

Система імітаційного моделювання мереж COMNETIII дозволяє точно прогнозувати продуктивність локальних, глобальних і корпоративних мереж. Система працює в середовищі Windows і Unix.

COMNETIII пропонує використовувати простий і інтуїтивно зрозумілий спосіб конструювання моделі мережі, заснований на застосуванні готових базових блоків, відповідних добре знайомим мережевих пристроїв, таким як комп'ютери, маршрутизатори, комутатори, мультиплексори і канали зв'язку.

Користувач застосовує техніку drag-and-drop для графічного зображення модельованої мережі з бібліотечних елементів. Потім система COMNETIII виконує детальне моделювання отриманої мережі, відображаючи результати динамічно у вигляді наочної мультиплікації результуючого трафіку (рис. 1.5).

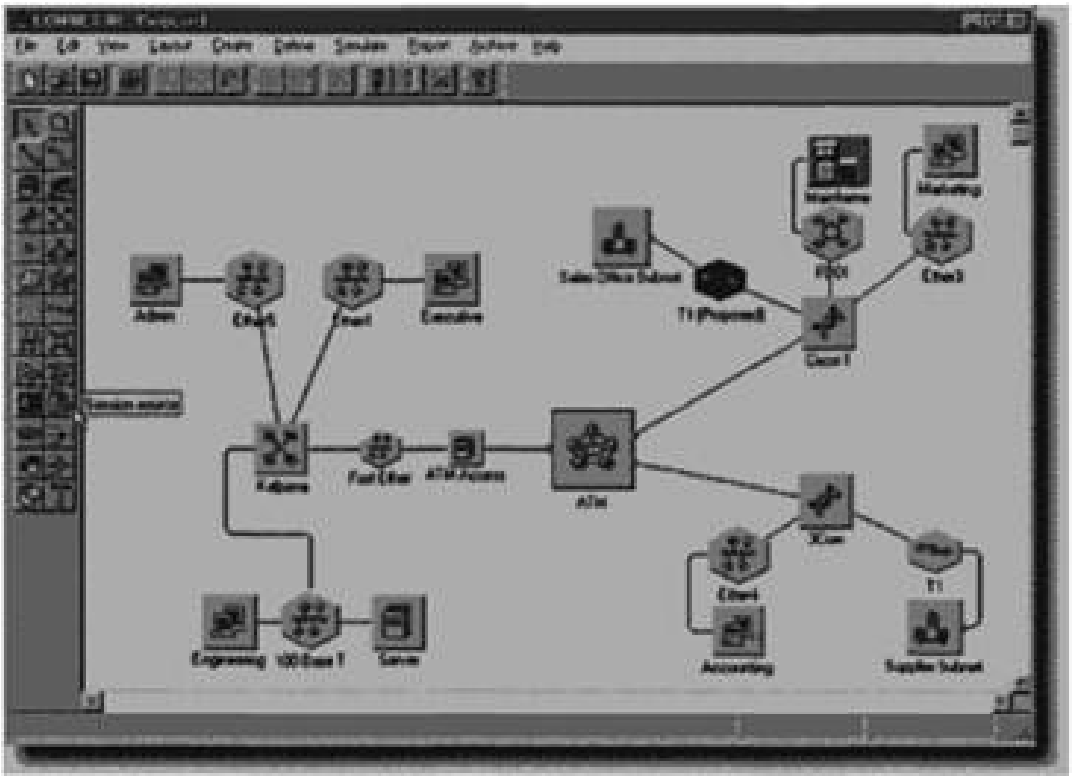

Рис. 1.5. Екранна форма головного вікна COMNETIII

Після закінчення моделювання користувач отримує в своє розпорядження наступні характеристики продуктивності мережі:

— Прогнозовані затримки між кінцевими і проміжними вузлами мережі, пропускні спроможності каналів, коефіцієнти використання сегментів, буферів і процесорів.

— Піки і спади трафіку як функцію часу, а не як усереднені значення.

— Джерела затримок і вузьких місць мережі.

У COMNETIII моделюється не тільки взаємодія комп'ютерів по мережі, але і процес поділу процесора кожного комп'ютера між його додатками. Робота програми моделюється за допомогою команд декількох типів, в тому числі команд обробки даних, відправки та читання повідомлень, читання і запису даних в файл, встановлення сесій і припинення програми до отримання повідомлень.

Канали зв'язку моделюються шляхом завдання їх типу, а також двох параметрів - пропускної здатності і вноситься затримки поширення. Одиницею переданих по каналу даних є кадр. Пакети при передачі по каналах сегментируются на кадри. Кожен канал характеризується: мінімальним і максимальним розміром кадру, накладними витратами на кадр і інтенсивністю помилок в кадрах.

COMNETIII дозволяє при моделюванні задавати форму звіту про результати для кожного окремого елемента моделі. Звіт генерується кожен раз при запуску певної моделі. Звіт представлений в стандартній текстовій формі, і його легко можна роздрукувати на будь-якому принтері. Можна задати генерацію кількох звітів різного типу для кожного елемента мережі.

Існують інші способи отримання статистичних результатів прогону моделі, крім звітів. У COMNETIII є кнопки Statistics, за допомогою яких можна включити збір статистики для кожного типу елемента моделі - вузлів, каналів, джерел графіка, маршрутизаторів, комутаторів і т. Д. Монітор статистики кожного елемента можна встановити для збору тільки базових статистичних параметрів (мінімум, максимум , середнє значення і дисперсія) або ж збору даних в тимчасовому масштабі для побудови графіків. Якщо результати

спостережень збережені у файлі для подальшої побудови графіків і аналізу, то можливо також побудова гістограм і процентних показників.

Будівник звітів, монітор статистики, апарат візуалізації є конкурентними і відмітними рисами цієї розробки.

1.6 Програмний комплекс IxChariot

IxChariot - це інструмент на базі програмного забезпечення для оцінки мереж, використовуваний для виміру ключових функціональних характеристик, таких як пропускна спроможність, година затримки, втрата пакетів, jitter, MOS для VoIP і MDI для відео в реальних умовах, і використовується щодня провідними компаніями і випробувальними лабораторіями для атестації і сертифікації новітніх мережевих пристроїв[7].

Виміри робочих характеристик проводяться шляхом передачі реальних потоків даних між прибудовами, підключеними до мережі. IxChariot емулює різні типи розподілених застосувань, збирає і аналізує отримані результати. Кінцеві точки IxChariot генерують трафік, використовуючи ті ж методи, що і будь-яке мережеве застосування, дозволяючи виміряти кожен елемент в тракті передачі даних.

Статистичної інформації тільки по мережевому рівню недостатньо для того, щоб передбачити робочі характеристики прикладного рівня в корпоративних мережах і широкосмугових транспортних мережах. Наприклад, розуміння впливу втрати пакетів має значення тільки в контексті конкретних мережевих застосувань. IxChariot® можна використати для досягнення максимальної продуктивності мережі і пристроїв.

Wireshark (раніше — Ethereal) — програма-аналізатор трафіку для комп'ютерних мереж Ethernet і деяких інших. Має графічний інтерфейс призначений для користувача. Функціональність, яку надає Wireshark, дуже схожа з можливостями програми tcpdump, проте Wireshark має графічний призначений для користувача інтерфейс і значно більше можливостей по

сортуванню і фільтрації інформації. Програма дозволяє користувачеві переглядати увесь трафік, що проходить по мережі, в режимі реального часу, переводячи мережеву карту в нерозбірливий режим.(англ. promiscuous mode)

Програма поширюється під вільною ліцензією GNU GPL і використовує для формування графічного інтерфейсу кроссплатформенную бібліотеку GTK+ (планується перехід на Qt) Існують версії для більшості типів UNIX, у тому числі Linux, Solaris, FreeBSD, NetBSD, OpenBSD, Mac OS X, а також для Windows.

Wireshark — це застосування, яке «знає» структуру найрізноманітніших мережевих протоколів, і тому дозволяє розібрати мережевий пакет, відображаючи значення кожного поля протоколу будь-якого рівня. Оскільки для захоплення пакетів використовується WinPcap, існує можливість захоплення даних тільки з тих мереж, які підтримуються цією бібліотекою. Проте, Wireshark уміє працювати з множиною форматів вхідних даних, відповідно, можна відкривати файли даних, захоплених іншими програмами, що розширює можливості захоплення.

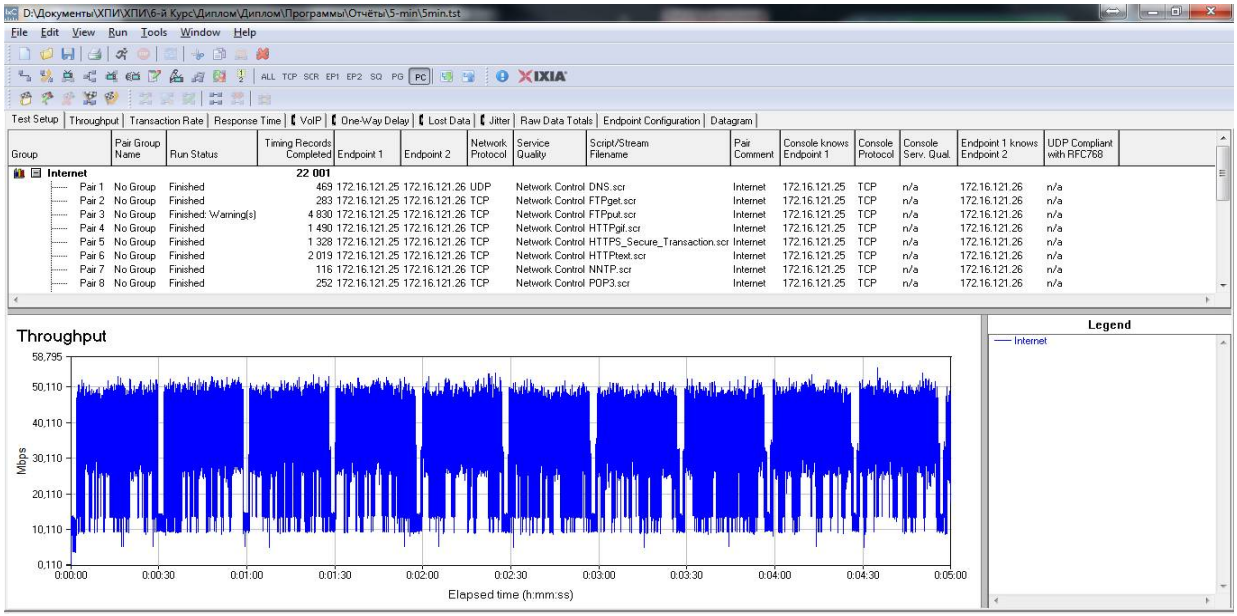

Рис. 1.6. Екранна форма головного вікна IxChariot

AutoSignal 1.7 є першим і єдина програма, яка повністю автоматизує процес аналізу сигналів. Це надає дослідникам право швидко знайти компоненти складних сигналів, які зазвичай вимагають широкого програмування і математичних функцій. Це повною мірою її графічного інтуїтивно зрозумілий інтерфейс користувача, щоб спростити усі аспекти роботи, з імпорту даних для виводу результатів.

Авторегресійних лінійні моделі пропонують надійні моделі, які можуть швидко обробляти менше наборів даних, які ШПФ (швидке перетворення Фур'є) не може точно аналізувати. З AutoSignal, ви можете також відновити компоненти сигналу на підставі потужності - компонент може бути синусоїдальним, меандр, пилкоподібний або ангармонічний малюнок. Вона дозволяє побачити повну картину частотною простору з використанням бібліотеки шести методів Фур'є спектру з повною гнучкістю.

Це дає вам вибір з трьох регульованих матерій сплесків : Морле, Павла і гаус похідних - в реальних і складних формах для оптимізації результатів локалізації.

Учені і інженери можуть виконувати комплексний аналіз сигналу без програмування, вибравши пункти меню, які визначають, як комп'ютер аналізуватиме і представлятиме дані. Вибір етапу обробки або алгоритму викликає до програмного забезпечення для подальших меню або вікон, так що користувачі можуть адаптувати продуктивність на їх потребі. Після того, як користувач робить вибір обробки, програмне забезпечення відразу представляє результати обробки в 2 - D або 3 - D виді

Вбудовані процедури спектрального аналізу включають:

— ШПФ;

— авторегресійний ;

— ковзаюче середнє;

— АРМА ;

— Комплекс експоненціальне моделювання;

— Методи мінімальної дисперсії;

— Ейген аналіз оцінки частоти і сплесків.

1.7 Висновки

В табл. 1.1 приведены характеристики нескольких популярных программ имитационного моделирования разного класса - от простых программ к мощным системам, которые включают библиотеки большинства имеющихся на рынке коммуникационных устройств и обеспечивают возможность значительной степени автоматизации исследования рассматриваемой сети.

Таблица 1.1 - Программы имитационного моделирования сетей

Компания, и название программы	Назначение
American HYTech, Prophesy	Оценивание производительности при работе с текстовыми и графическими данными по отдельным сегментам и сети в целом.
CACI Product, COMNET III	Моделирует сети X.25, ATM, Frame Relay, связи LAN-WAN, SNA, DECnet, протоколы OSPF, RIP. Доступ CSMA/CD, FDDI и др. Устроенная библиотека маршрутизаторов 3COM, Cisco, DEC, HP, Wellfleat.
Make System, NetMaker XA	Проверка данных о топологии сети; импорт информации о трафике, получаемой в реальном времени.
NetMagic System, StressMagic	Поддержка стандартных тестов измерения производительности; имитация пиковой нагрузки на файл-сервер.
Network Analysis Center, MIND	Средство проектирования, оптимизации сети, содержит данные о стоимости типичных

	конфигураций с возможностью точного оценивания производительности.
Network Design and Analysis Group, AutoNet/Designer	Определение оптимального расположения концентратора в ГС, возможность оценки экономии средств за счет снижения тарифной платы, изменения поставщика услуг и восстановления оборудования; сравнение вариантов связи через ближайшую и оптимальную точку доступа, а также через мост и местную телефонную сеть.
Network Design and Analysis Group, AutoNet/ MeshNET	Моделирование полосы пропускания и оптимизация затрат на организацию ГС путем имитации поврежденных линий, поддержка тарифной сетки компаний AT & T, Sprint, WiTel, Bell.
Network Design and Analysis Group, AutoNet/Performance-1	Моделирование производительности иерархических сетей путем анализа чувствительности к продолжительности задержки, времени ответа, а также узких мест в структуре сети.
Network Design and Analysis Group, AutoNet/Performance-3	Моделирование производительности многопротокольных объединений локальных и глобальных сетей; оценивание задержек в очередях, прогнозирование времени ответа, а также узких мест в структуре сети; учет реальных данных о трафике, что поступают от сетевых анализаторов.
System& Networks, BONES	Анализ влияния добавлений клиент-сервер и новые технологии на работу сети.
MIL3, Opnet	Имеет библиотеку разных сетевых устройств, поддерживает анимацию, генерирует карту сети, моделирует полосу пропускания.

Рекомендації щодо обирання програмного засобу моделювання інфокомунікаційних мереж залежать від потреб задачі користувача та його сподівань щодо необхідного часу.

Зваживши мінуси та плюси оглянутих програмних засобів рекомендовано у рамках відповідного курсу для практичного вивчення обрати програмний засіб OmNet++.

Контрольні питання.

1. Які на вашу думку повинні бути критерії та якості програмних засобів моделювання інфокомунікаційних мереж, так які з них найбільш впливові, якщо використовувати їх у навчальному процесі?
2. Які плюси використання C++ у якості системної мови програмного комплексу Network Simulator (NS-2)
3. Призначення програмного комплексу OPNET. Які моделі протоколів він здатний моделювати?
4. Який основний набір модулів програмного комплексу OMNeT++?
5. Які основні мови програмування використовуються у OMNeT++?
6 Призначення симулятора javaNetSim та його архітектура.
7. Які результати роботи програмного комплексу COMNETIII отримує користувач?
8. Структура модулів програмного комплексу IxChariot. Вбудовані процедури спектрального аналізу IxChariot.

2. ПОЧАТОК РОБОТИ З OMNET++

2.1 Призначення OMNeT++

OMNeT++ – це здатний розширюватись модульний комплекс бібліотек із фреймворком, заснований на компонентах C++, призначений для побудови, у першу чергу, моделей мереж. "Мережа" у більш широкому розумінні містить у собі дротові (wired) і бездротові (wireless) мережі зв'язку, мережі на кристалі (on-chip), мережі обслуговування (queueing networks) і т.д. Специфічні доменні функції, наприклад, підтримка сенсорних мереж (sensor networks), бездротових однорангових мереж (ad-hoc networks), Інтернет протоколів, моделювання продуктивності, моделювання фотонних мереж (photonic networks), та інші, забезпечуються в рамках набору фреймворків моделей, розроблених як самостійні проекти. OMNeT++ надає інтегроване середовище розробки на базі Eclipse IDE, графічного середовище виконання (runtime environment) і безліч інших інструментів. Так само мається розширення для моделювання в реальному часі (real-time simulation), емуляції мережі, інтеграції з базами даних, інтеграції з мовою Systemc і ряд інших функцій.

Хоча OMNeT++ сам по собі не є мережним симулятором, він одержав широку популярність у користувачів у науковому співтоваристві й промислових підприємствах у якості мережної платформи для моделювання.

OMNeT++ надає архітектуру компонентів для моделей. Компоненти (модулі) розроблені на C++, а потім зібрані в більші компоненти й моделі з використанням мови високого рівня (NED). Можливість повторного використання моделей поставляється безкоштовно. OMNeT++ має широку підтримку графічного інтерфейсу, і, завдяки своїй модульній архітектурі, ядро моделювання й моделі можуть бути легко вбудовані в розроблювальні користувачами додатки.

Модульний комплекс OMNeT++ складається з наступних компонентів:

- бібліотека ядра моделювання;
- мова опису топології NED (Network Description);
- інтегроване середовище розробки OMNeT++ IDE, заснована на платформі Eclipse;
- графічний інтерфейс користувача (GUI) для виконання симуляції (середовище Tkenv);
- користувацький інтерфейс командної строки для виконання моделювання (середовище Cmdenv);

– інші утиліти (інструмент створення файлів makefile і т.п.);
– документація, приклади виконання симуляцій і т.п.

OMNeT++ працює під керуванням операційних систем сімейств Windows, Linux, Mac OS X і Unix-Подібних.

Скачати настановний пакет OMNeT++ можна із сайту www.omnetpp.org

2.2 Установка OMNeT++

Необхідно розпакувати настановний пакет OMNeT++ в обрану папку на диску, де він буде постійно розміщений. Запуск і настроювання комплексу здійснюється через консоль, яка у свою чергу запускається з файлу mingwenv.cmd.

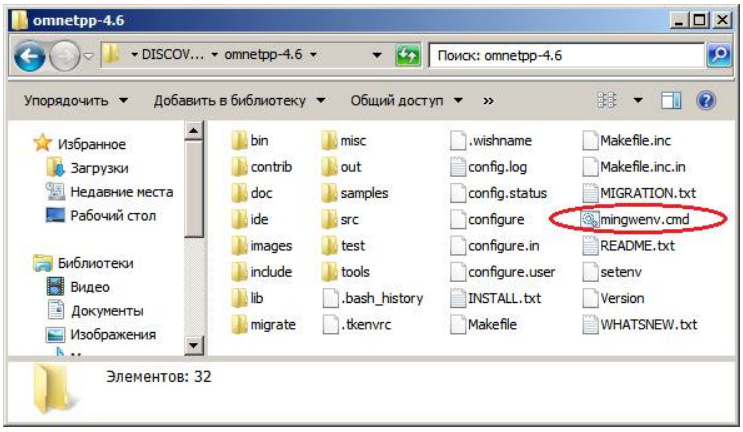

Рис.1 – Структура каталогу OMNeT++

Перед першим запуском комплексу, необхідно виконати його конфігурування й установку.

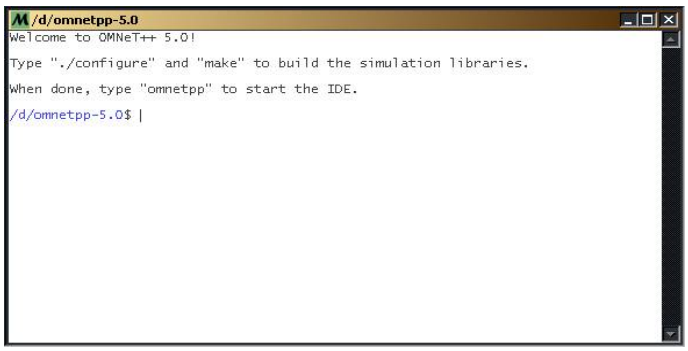

Рис.2 – Консоль для конфігурування й старту системи

Команда ./configure виконає конфігурування системи. Після цього треба виконати команду make. Виконання обох команд забирає тривалий час. Для запуску використовується команда omnetpp.

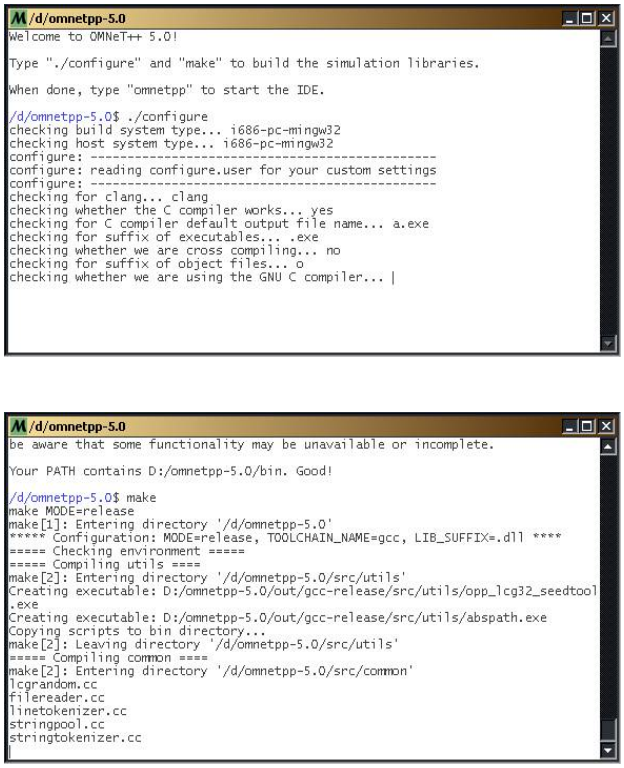

Рис.3 – Виконання команд настроювання й установки в консолі

У запропонованому діалоговому вікні треба вибрати робочу папку для зберігання проектів.

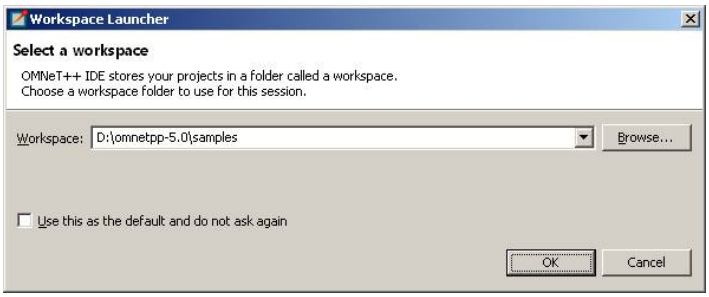

Рис.4 – Діалог вибору папки для зберігання проектів

Після запуску середовища OMNeT++ буде запропоновано встановити із сайту розроблювачів фреймворк для моделювання мереж і приклади проектів.

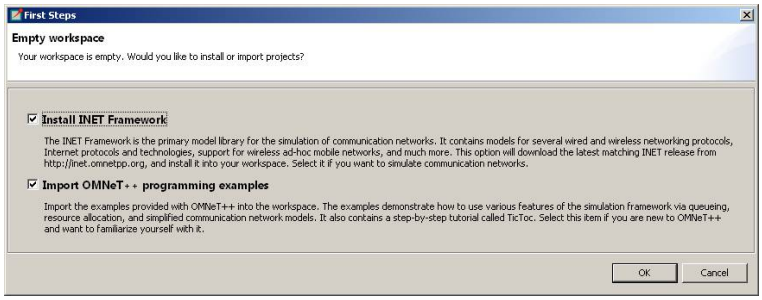

Рис.5 – Завантаження фреймворку й прикладів

Після цього можна приступати до створення власних проектів.

2.3 Висновки.

Таким чином у розділі було розглянуто структуру та призначення загальних модулів Omnet++. Було описано різні середовища виконання моделювання. Було наглядно показано налаштування, конфігурування та встановлення програмного комплексу Omnet++. Було розроблено перший проект моделі, побудована перша примітивна мережа засобами візуального розташування об’єктів на мережі та шляхом написання програмного коду мовою NED.

Запуск першого проекту мережі надав результати моделювання, було проведено аналіз результатів для різних апріорних даних.

Було показано механізми налаштування апріорних даних проекту та засоби аналізу результатів. Було показано та проведено аналіз графіків черг, шляхів пакетів та фільтрації пакетів у часі.

Контрольні питання.

1.Призначення Omnet++. Які модулі та компоненти включає комплекс Omnet++?
2. Етапи налаштування, компіляції та встановлення Omnet++.
3. Приклад створення нового проекту.
4. Приклад створення NED файлу
5. Які типи підмодулів доступні з палітри компонентів?
6 Принципи конфігурування моделі. Структура конфігураційного файлу.
7. Логування процесу імітації. Типи лог-файлів.

8. Конфігурування запуску моделі. Які середовища запуску моделей використовує Omnet++?
9. Які файли створюються після завершення сеансу імітації в папці проекту?
10. Аналіз результатів моделювання. Типи файлів аналізів.
11. Яку інформацію надає графік послідовностей подій?
12 Яку інформацію надає часовий графік з нелінійним часом?

3. СТВОРЕННЯ ПЕРШОЇ ІМІТАЦІЙНОЇ МОДЕЛІ

3.1 Створення нового проекту

Розглянемо, як створювати, конфігурувати, запускати й аналізувати імітаційні моделі в інтегрованому середовищі розробки Omnet IDE.

Середовище розробки OMNeT++ IDE побудована на технології Eclipse. Тому необхідно вибрати перспективу OMNeT++ (Omnet Perspective) – специфічний вид інтерфейсу, який перемикає робоче місце розробника (workbench) зі стиля Eclipse до розмітки (layout), яку оптимізовано під OMNeT++, і, також, додає специфічні пункти меню. Для цього необхідно вибрати пункт меню *Window->Open Perspective->Simulation* (рис.6).

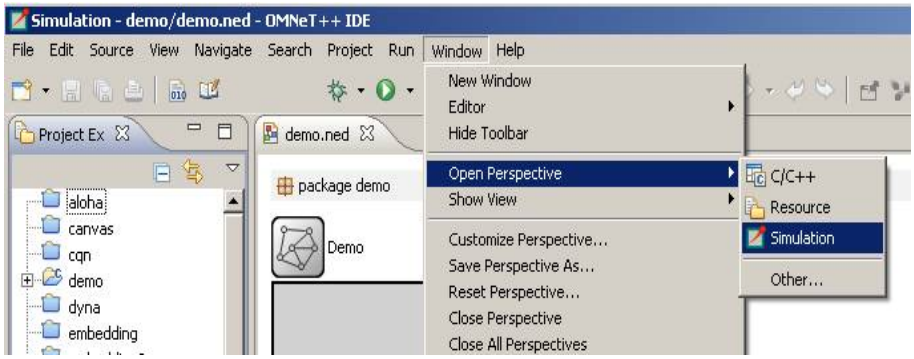

Рис.6 – Зміна перспективи

Щоб створити новий проект імітаційної моделі OMNeT++ використовується OMNeT++ Project wizard. Створений проект буде містити в собі робочі файли. Вибираємо пункт меню *File->New->OMNeT++ Project* (рис.7).

Рис.7 – Запуск OMNeT++ Project wizard

Необхідно ввести ім'я проекту, наприклад, назвемо проект demo. Потім натискаємо кнопку *Finish* (мал.8, мал.9). У списку проектів (Project explorer) з'явиться новий проект.

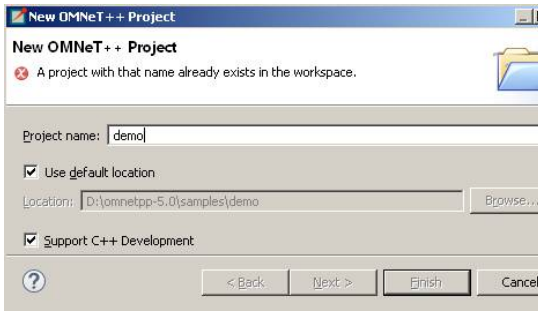

Рис.8 – Створення проекту

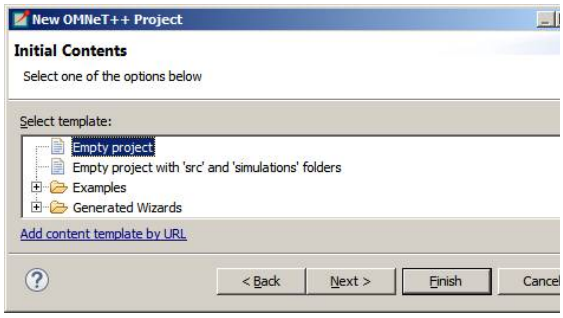

Рис.9 – Вибір шаблону проекту

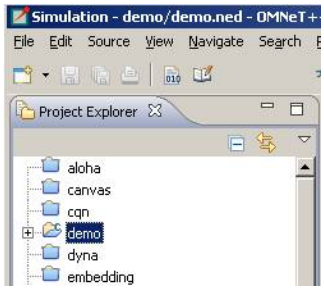

Рис.10 – Відображення створеного проекту demo

У рамках даного проекту побудуємо імітаційну модель мережі обслуговування (queueing network). Мережа, яку моделюємо, буде зібрана з компонентів, визначених у вже існуючому проекті queueinglib. Тому необхідно, додати його до залежностей розроблювального проекту demo від queueinglib у властивостях проекту (Project Properties). Для цього треба відкрити контекстне меню проекту, нажати правою кнопкою миші на назві проекту, вибрати пункт меню *Properties*. У діалозі властивостей вибираємо закладку *Project References* і відзначаємо в списку галочкою проект queueinglib.

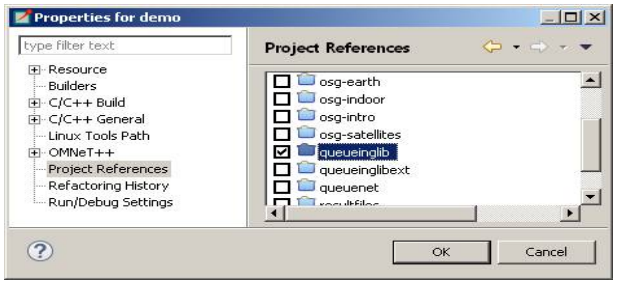

Рис.11 – Залежності проектів

3.2 Створення NED файлу

На наступному етапі необхідно створити NED файл (файл опису мережі), використовуючи NED File wizard. Вибираємо з головного меню пункт *File->New->Network Description File (NED)*.

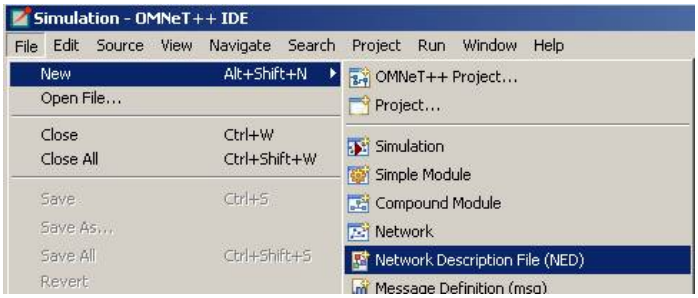

Рис.12 – Пункт меню для створення NED файлу

У діалозі, який відкрився (рис.13) уводимо ім'я файлу, наприклад, назвемо його demo.ned. Натискаємо кнопку *Next*, відкривається діалог вибору початкового шаблону моделі (рис.14). Виберемо в якості початкового шаблону моделі NED файл із одним елементом *NED file with one item*. Натискаємо кнопку *Next*, вибираємо в якості єдиного елемента мережу (рис.15), пункт *Network*. Візуалізація отриманої порожньої моделі наведена на рис.16.

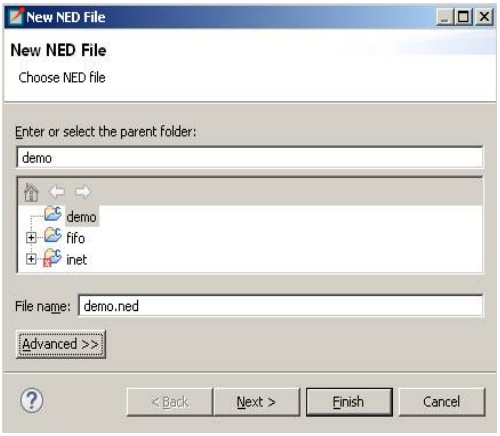

Рис.13 – Уведення назви NED файлу в діалозі

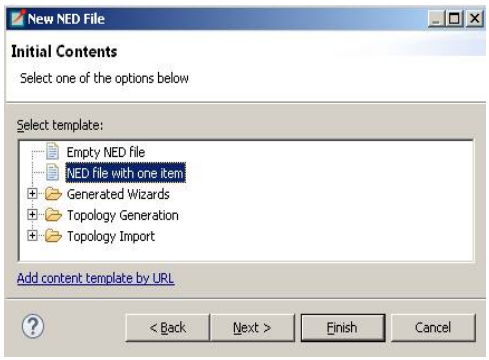

Рис. 14 – Вибір початкового шаблону моделі

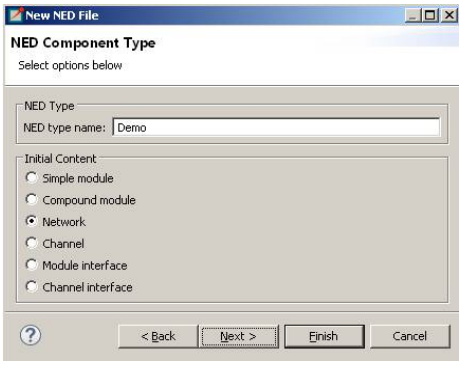

Рис.15 – Вибір компонента «Мережа» у якості стартового компонента

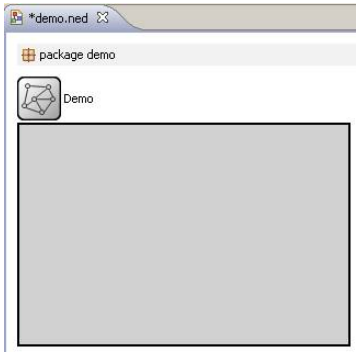

Рис.16 – Отримана порожня мережа у файлі моделі demo.ned

Побудуємо тепер закриту мережу обслуговування (closed queuing network) з єдиним вузлом-джерелом (source node) і трьома чергами із вбудованим сервером, поєднанні кільцем.

Типи підмодулів доступні з палітри (palette) у правій частині графічного редактора. Будемо використовувати підмодулі із проекту queueinglib. Якщо підмодулі з queueinglib не відобразилися в палітрі, то треба обрати в списку проектів проект queueinglib і зайти в нього. Додамо три черги із вбудованим сервером до мережі. Для цього з палітри підмодулів виберемо компонент Queue, додамо його до мережі demo і задамо ім'я для кожної черги, як показано на рис.17. Потім установимо джерело Source з палітри компонентів (рис.18).

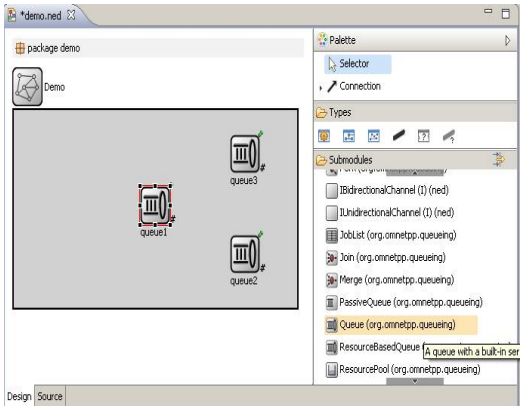

Рис. 17 – Додання трьох черг

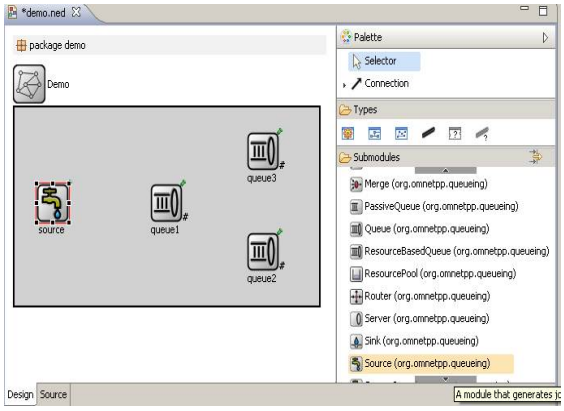

Рис. 18 – Додання джерела

Тепер з'єднаємо підмодулі, використовуючи засіб з'єднання модулів Connection (рис.19). З'єднуючи підмодулі, у меню, що випадає, необхідно вибирати входи й виходи (Gates) підмодулів.

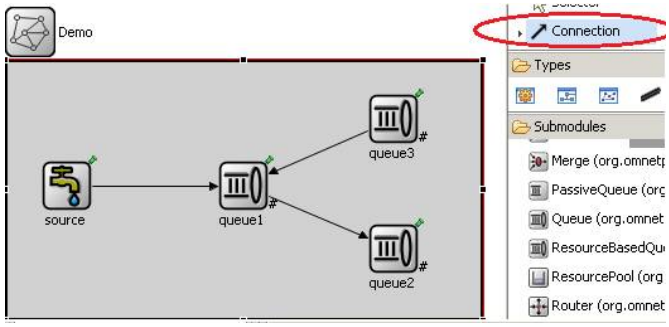

Рис. 19 – З'єднання підмодулів

Мережу також можна редагувати в текстовому редакторі вмісту файлу (Content Assist), для цього треба перемкнутися із закладки *Design* до закладки *Source*. Допишемо рядок у редакторі, який додасть з'єднання між другою й третьою чергою, як показано на рис.20.

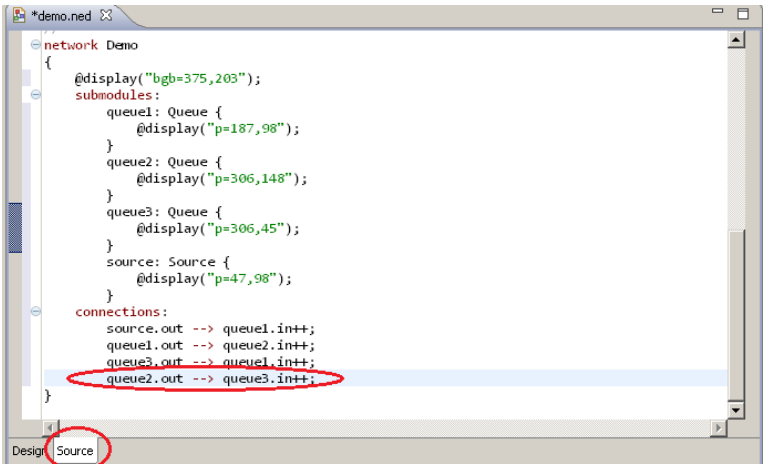

Рис. 20 – Додавання з'єднання між підмодулями в редакторі

3.3 Конфігурування моделі

Перед запуском імітаційної моделі мережі, її слід спочатку налаштувати. Для цього необхідно створити INI файл (Initialization File) і вказати в ньому параметри моделі. Використовуємо майстер налаштування INI файлів (Ini File Wizard), щоб створити файл. Вибираємо в контекстному меню проекту пункт *New->Initialization File* (рис.21).

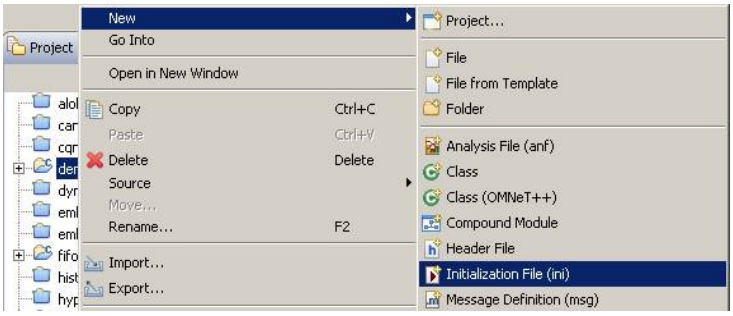

Рис. 21 – Меню створення файлу параметрів моделі

У наступних діалогах вводиться ім'я файлу настроювань (рис.22) і шаблон (рис.23).

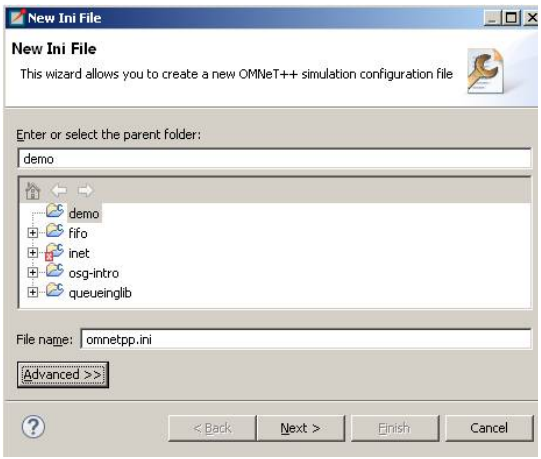

Рис. 22 – Введення назви конфігураційного файлу

Рис. 23 – Створюється порожній конфігураційний файл

Вибираємо мережу, для якої створюється конфігураційний файл, тобто виберемо мережу поточного проекту. Кнопкою *Browse* необхідно відкрити список доступних моделей (рис.24) і вибрати мережу поточного проекту demo.Demo.

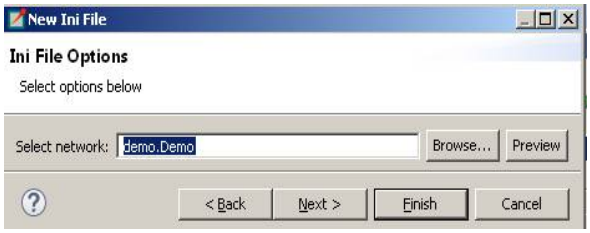

Рис. 24 – Вибір мережі

Конфігураційний файл являє собою текстовий файл, який можна редагувати як у текстовому режимі, так і з використанням спеціальних екранних форм.

Далі необхідно встановити параметри моделі, значення яких не задані за замовчуванням. У закладці *Parameters* (рис.25) натискаємо кнопку *Add*, у діалозі, що відкрився, відображений список невстановлених параметрів, які необхідно додати в конфігураційний файл. Спочатку встановлюємо параметри interarrivaltime і кількість виконуваних робіт numjobs у підмодулі-джерелі (Source) (рис.26).

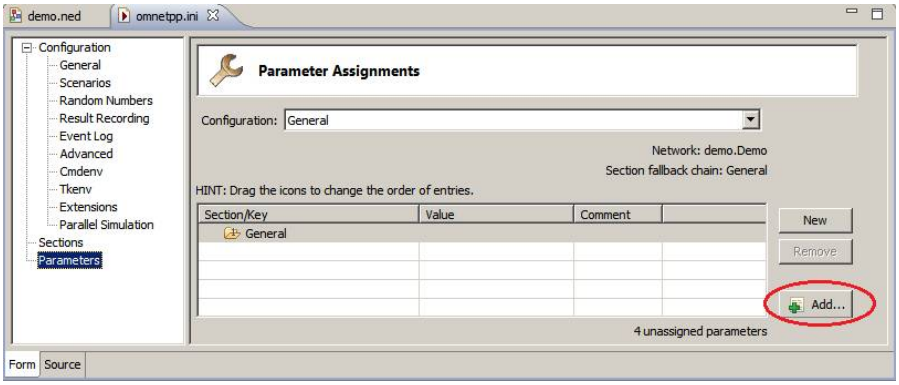

Рис.25 – Форма введення параметрів моделі

Встановимо джерелу (Source) параметр interarrivaltime рівним нулю, цим самим вкажемо моделі, що всі роботи із джерела без затримок миттєво надходять до мережі обслуговування (queuing network). Необхідно, щоб модель працювала з варіантами двох конфігурацій: з 30, а потім з 60 початковими роботами. Спеціальний синтаксис ${…} дозволяє задати такі умови безпосередньо в конфігураційному файлі ini. Задамо параметр numjobs рівним ${jobs=30,60} (рис.27).

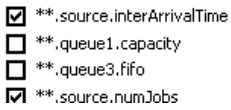

Рис.26 – Параметри моделі, які додаються в модель

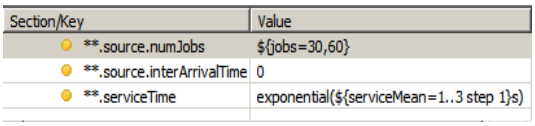

Рис.27 – Список параметрів, що задаються, моделі в конфігурації

Далі задамо час обслуговування (service time) для всіх черг (queues) у системі. Необхідно задати тільки ті параметри, у яких немає значення за замовчуванням, заданих в NED файлі (рис.27).

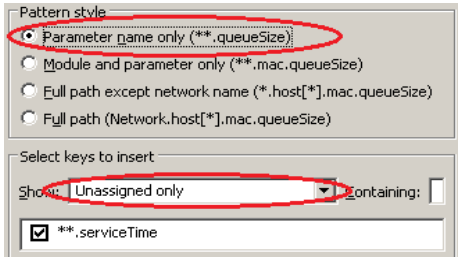

Рис.28 – Додавання параметра, що задає час обслуговування

Необхідно випробувати модель для різних варіантів часу обслуговування, який розподілений експоненційно (exponential distribution) із середнім значенням (mean) рівним 1, 2 і 3. Для цього встановимо servicetime як експоненційно розподілений параметр із мінливим середнім значенням у секундах, рівний exponential(${servicemean=1..3 step 1}s).

Можна було б задати виконання імітаційного моделювання, з використанням кожної конфігурації кілька раз із різними початковими значеннями, але кількість ітерацій, що задається параметрами numjobs і interarrivaltime у поточному проекті, дає достатню кількість запусків для демонстрації.

Наступним кроком необхідно включити логування (logfile generation), щоб аналізувати ітерації між модулями надалі. Для цього вибираємо закладку *Event Log* (рис.29).

Далі треба задати тривалість виконання імітаційного моделювання. Для цього вибираємо закладку *General* і задаємо параметр *Simulation time limit* рівним 200s (рис.30). Потім задамо інтервали часу, через які необхідно видавати на консоль статус моделювання. Для цього виберемо закладку *Cmdenv* і встановимо параметру *Status frequency* значення 2s (рис.31).

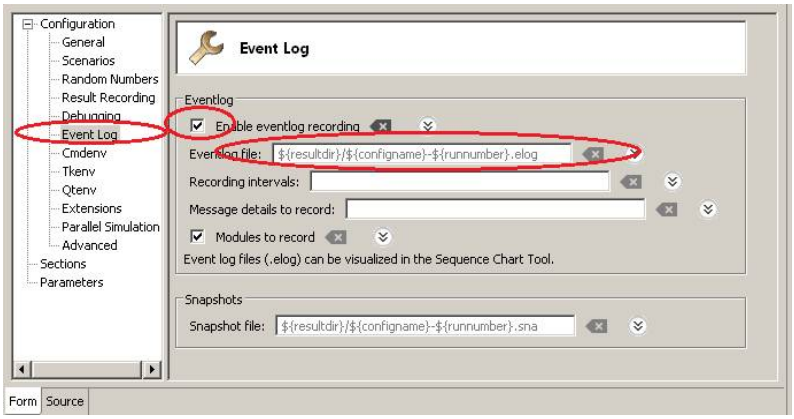

Рис.29 – Включення логування

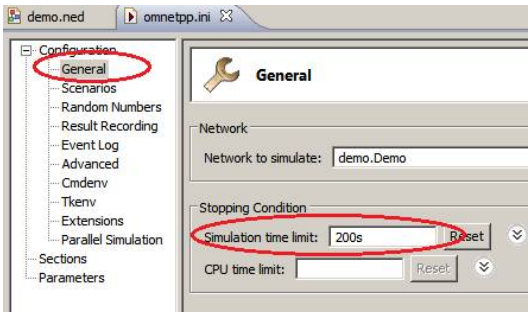

Рис.30 – Введення тривалості моделювання

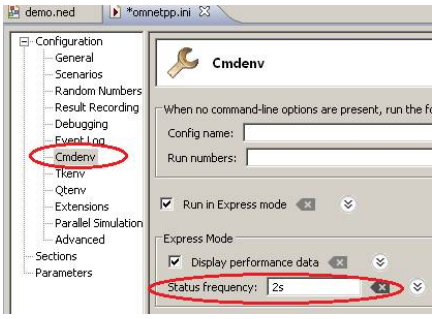

Рис.31 – Введення інтервалів між виводами статусу моделі

Перемкнувшись на закладку *Source*, можна побачити конфігураційний файл у текстом виді, представленому на рис.32.

Рис.32 – Текстовий вид конфігураційного файлу

3.4 Конфігурування запуску моделі

Таким чином, отримана модель мережі описана в NED файлі, а в INI файлі задані відкриті параметри системи. На наступному кроці необхідно запустити імітаційну модель із інтегрованого середовища розробки (IDE). Для цього необхідно створити конфігурацію запуску (Launch Configuration), вибираємо пункт головного меню програми *Run->Run Configurations…* (рис.33), відкриваємо закладку *OMNeT++ Simulation* (рис.34).

Рис.33 – Пункт меню настроювання конфігурації запуску

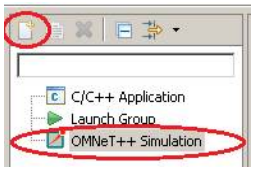

Рис.34 – Настроювання конфігурації запуску

На рис.35 представлена форма для завдання параметрів конфігурації. У полі *Name* задається назва конфігурації. Робочим каталогом (Working directory) обраний за замовчуванням каталог /demo. У розділі *Simulation* задаємо для опції *Executable* значення opp_run. У параметр *Ini file(s)* уписуємо назву конфігураційного файлу omnetpp.ini, який був створений у рамках даного проекту. У створеному конфігураційному файлі була задана тільки одна конфігурація General, її й вибираємо зі списку, що випадає, у *параметра Config* name.

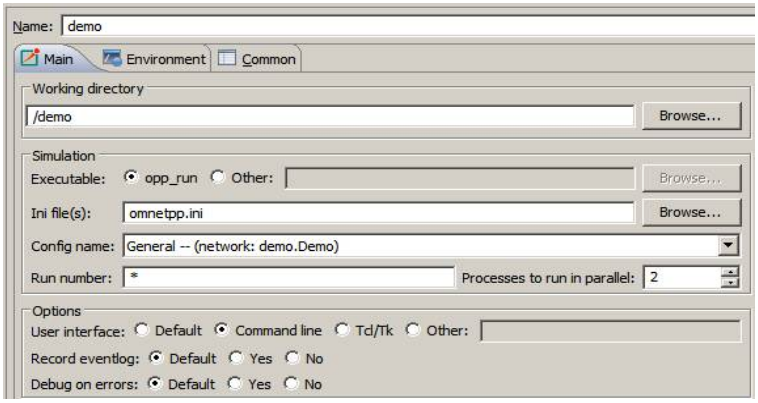

Рис.35 – Екранна форма настроювання конфігурації запуску

В INI файлі було визначено, що імітаційна модель буде запускатися з 30 і 60 роботами, а час обслуговування буде експоненційно розподілений з варіантами середніх значень – 1, 2 і 3. Таким чином, буде 6 різних запусків. Необхідно запустити кожний варіант, тому встановимо параметру *Run number* (кількість запусків) значення * (зірочка).

Параметр *Processes to run in parallel* відповідає за кількість процесів, виконуваних одночасно. Поставимо йому значення – 2.

Перейдемо в групу параметрів *Options*. У прикладі будемо використовувати оточення командного рядка для запуску додатка, для цього виберемо параметру *User interface* значення Cmdenv.

3.5 Запуск сеансу імітації

Завершивши настроювання можна приступити до виконання сеансу імітації, натискаємо кнопку *Run* і стежимо за прогресом виконання у погляді *Progress View* (рис.36).

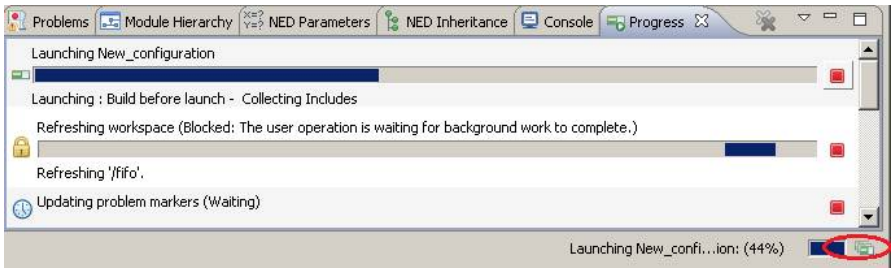

Рис.36 – Прогрес виконання сеансу імітації

Після завершення сеансу імітації в папці проекту будуть створені наступні файли: VEC і SCA містять статистику, записану імітаційною моделлю, LOG файл містить записи про кожне повідомлення, що посилає модель, текстові повідомлення дебагера й інші повідомлення. Записи LOG файлу можуть бути відображені в схемах послідовності (sequence charts).

3.6 Аналіз результатів моделювання

OMNeT++ пропонує спеціальні засоби аналізу результатів. Для цього створимо файл аналізів (Analysis) ANF, використовуючи спеціальні засоби. Виберемо пункт головного меню *File->New->Analysis File* (рис.36).

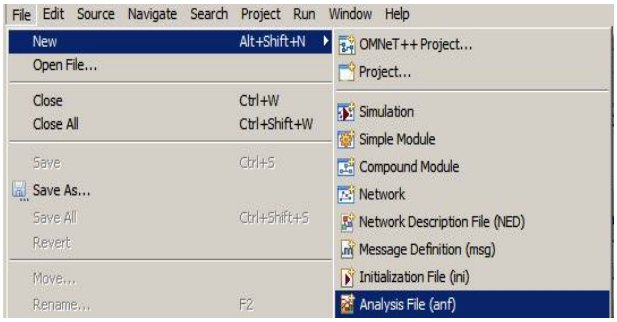

Рис. 36 – Створення файлу аналізів

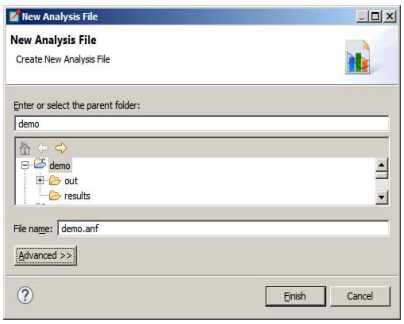

Рис.37 – Вибір папки проекту для файлу аналізів

Спершу слід додати всі сгенеровані файли результатів у файл аналізів. Можна задати точні імена файлів або ж перетягнути їх мишкою з навігатора, але найпростіше використовувати підстановочні групові символи (wildcards). Натискаємо кнопку *Wildcards*, задаємо шаблон вибору фалів - /demo/results/*.vec і /demo/results/*.sca, натискаємо кнопку *OK*.

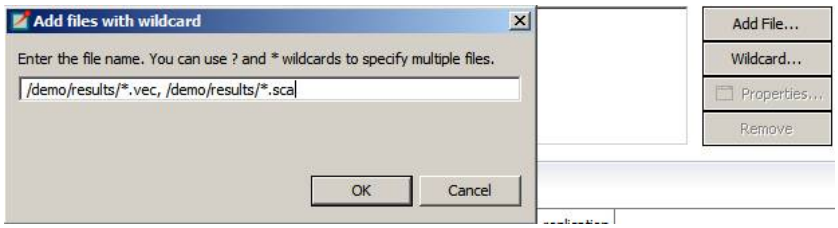

Рис.38 – Вказівка шаблонів вибору файлів

Редактор відобразить відповідні до шаблону файли нижче, по одному файлу векторів і файлу скалярів на кожний запуск моделі. Файли на згадку повністю не завантажуються, але їх уміст сканується програмою.

Щораз, коли запускається імітаційна модель, випробування (Run) одержує унікальний код (ID), який містить номер випробування, дату й час. Перша закладка групує результати по файлах, а усередині файлів по випробуваннях (рис.39).

Друга закладка відображає випробування (запуски), для обраного випробування можна переглянути файли даних.

Третя закладка представляє логічне угруповання випробувань (рис.40). Усі випробування, які були зроблені, належать до одного експеримента (experiment), який називається General по імені конфігурації, записаної в INI файлі. Назва експерименту, встановлене за замовчуванням, може бути перезаписане в INI файлі.

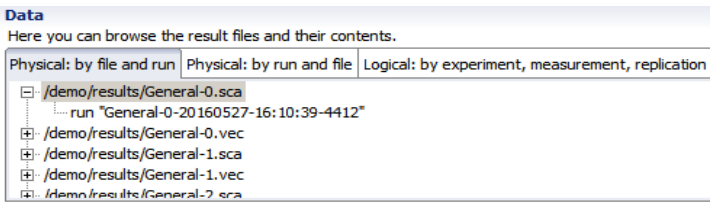

Рис.39 – Файли результатів експерименту

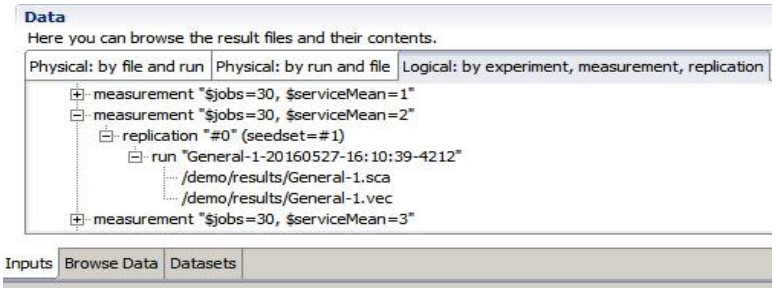

Рис.40 – Комбінації значень параметрів моделі

Кожний експеримент складається з декількох комбінацій значень параметрів моделі – вимірів (measurement), які задають різні варіанти роботи для однієї й тієї ж імітаційної моделі. Кожний змінний вхідний параметр моделі повторюється з різними значеннями для одержання статистично достовірних результатів у результаті виконання декількох репликацій (replication). Один екземпляр запущеної репліки одержує унікальний код (ID) запуску.

Перемкнувши програму на закладку перегляду даних *Browse Data*, будуть відображені вектори (vectors), записані під час виконання ітерацій імітаційного моделювання. Наприклад, становить інтерес, як довжина черги змінюється згодом. Для цього в списку, що випадає, фільтру статистик (Statistic Name Filter) встановимо значення queuelength. Далі, залишимо вектори значень тільки для четвертої ітерації. Для цього у фільтрі по номеру ітерації (runid filter) задамо значення General-4.

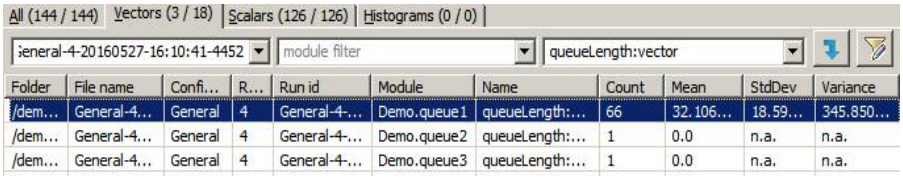

Рис.41 – Статистичні характеристики довжин черг

Відображені три вектори відповідають трьом чергам моделі мережі, по одному кожній черзі. Розмістимо їх на одному графіку (chart), натискаємо на панелі завдань кнопку *Plot* (рис.42).

Рис.42 – Кнопка створення графіка

У результаті виходить графік залежностей довжин черг від часу моделювання за весь період моделювання (рис.43). Змінити масштаб графіка можна, використовуючи кнопки Zoom Tool (Рис. 44).

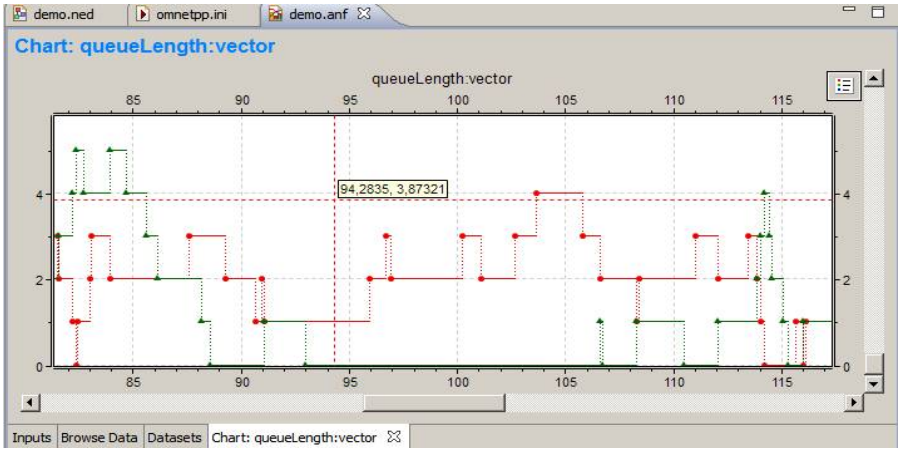

Рис.43 – Графік залежностей довжин черг від часу

Рис.44 – Кнопки зміни масштабу графіка Zoom Tool

Кликнувши правою кнопкою миші на графік й вибравши пункт редагування властивостей *Properties*, відкривається діалог модифікації зовнішнього вигляду графіків (рис.45).

Щоб задати усередюючу функцію (mean function) для одержання згладженого графіка кликнемо правою кнопкою миші на графіку, у меню, що відкрилося, *виберемо Apply*->Mean (рис.46).

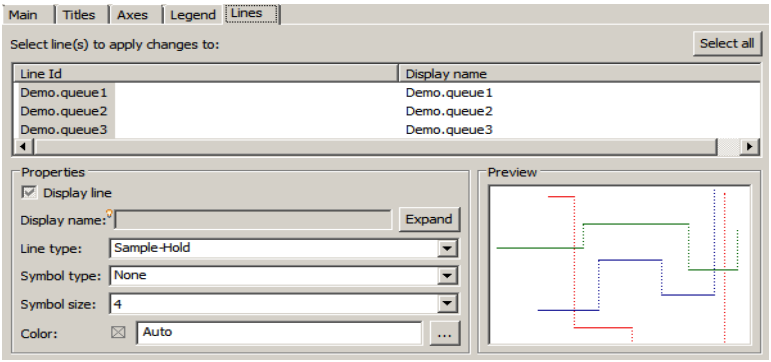

Рис. 45 – Редагування зовнішнього вигляду графіків

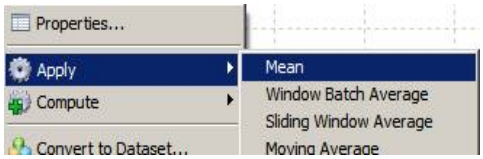

Рис. 46 – Вибір пункту меню, що створює усереднюючу функцію

Можна зберегти цей графік у вигляді автоматичної напередвстановленої інструкції, що представляє собою рецепт (recipe) по створенню графіка, таким чином, після перезапусків імітаційної моделі надалі побудований графік буде перестворюватися автоматично. Для цього кликнемо правою кнопкою миші на графіку, у меню, що з'явилося, виберемо *пункт Convert to* dataset (рис.47). У закладці *Datasets* відображаються отримані рецепти по створенню графіків і діаграм, які являють собою пророблені користувачем кроки зверху вниз.

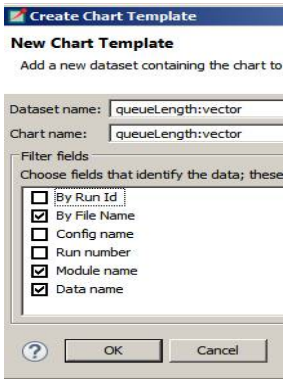

Рис.47 –Створення recipe

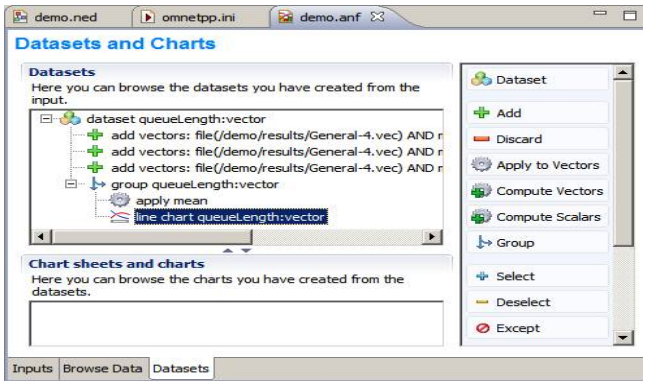

Рис. 48 – Список рецептів створення графіків

Зберігаючи файл аналізів (analysis) результатів, будуть записані тільки інструкції рецептів (recipe): який файл завантажувати, яку безліч даних вибирати, який графік необхідно будувати й відображати.

Щоб відкрити файл подій (log-файл) для перегляду графіка послідовностей подій (Sequence Chart) необхідно вибрати на закладці *Project Explorer* у папці проекту файл із розширенням «.elog», наприклад *demo->results->General-0.elog* (рис.49).

На екрані буде відображено графік послідовностей подій, який містить 60 початкових повідомлень, які вставлені в одну із черг (рис.50).

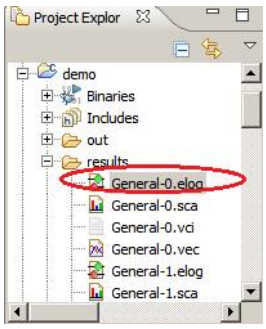

Рис. 49 – Вибір файлу подій

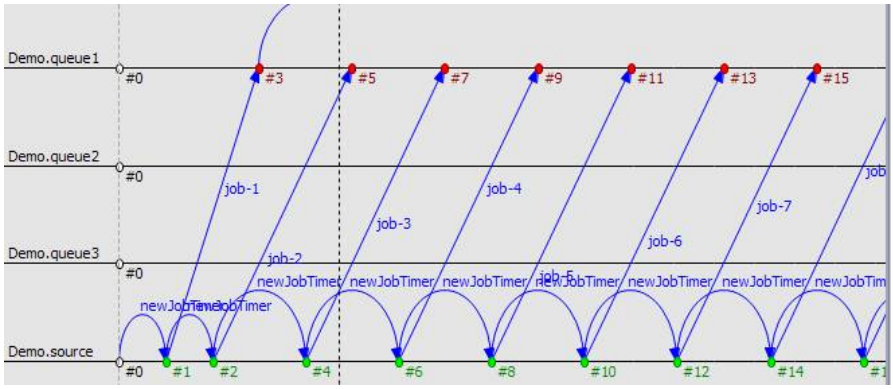

Рис. 50 – Графік послідовностей подій

Виберемо на графіку момент часу, коли було оброблено перше повідомлення. Кликнемо на ньому правою кнопкою мишки, у меню, що відкрилося, *виберемо Sending (cmessage) end-service (id=0) -> Go to Consequence* Event (рис.51), ця дія перенесе обрану часову позицію на наступну за поточним подію.

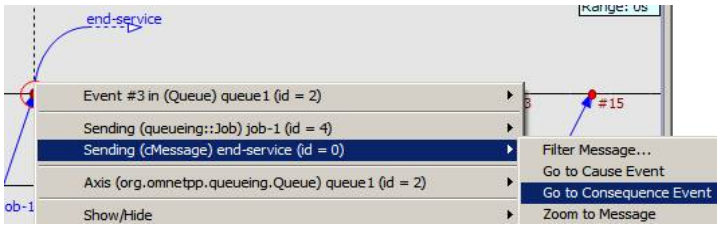

Рис. 51 – Графік послідовностей подій

Змінимо масштаб графіка, використовуючи Zoom Tool до потрібного розміру, як показано на рис. 52.

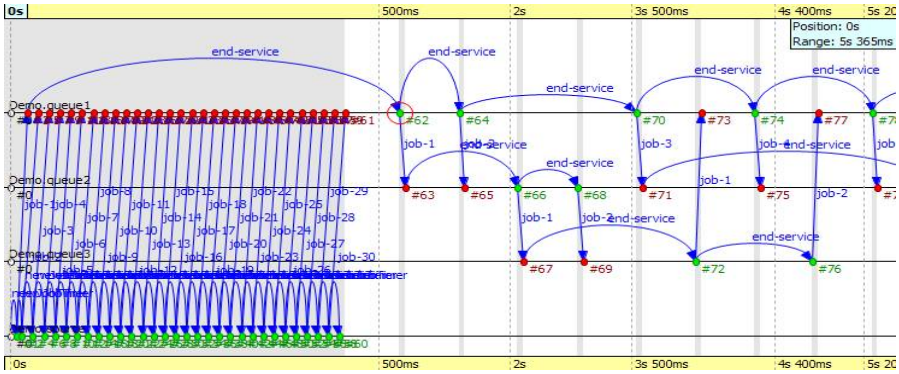

Рис. 52 – Часовий графік з нелінійним часом

У режимі нелінійної часової лінії, повідомлення в одній і тій же зафарбованій області мають те саме значення часу в рамках випробування імітаційної моделі. Якщо перемкнути графік у режим лінійного часу буде видне, що початкові повідомлення починають відсилатися в нульовий момент часу, а зафарбовані області редукуються у вертикальну лінію. Для зміни режиму часової лінії треба нажати на панелі завдань кнопку *Time Line Mode* (рис.53).

Відфільтруємо для початку повідомлення, щоб побачити, як вони окремо циркулюють у замкненій мережі. Щоб задати фільтр повідомлень треба нажати на панелі завдань кнопку *Filter* (рис.54).

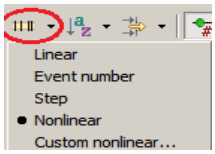

Рис. 53 – Меню зміни режиму часової лінії.

Рис. 54 – Меню зміни режиму часової лінії.

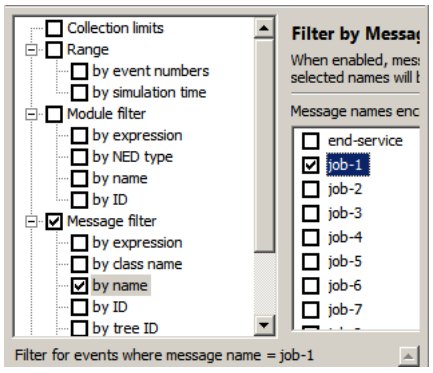

Рис. 55 – Меню зміни режиму часової лінії.

Перемкнувши графік до режиму нелінійного часу, видно, що обране повідомлення йде по колу кілька раз. Щоб показати, у якому місці повідомлення були використані заново шляхом повторного їхнього пересилання, виберемо в контекстному меню пункт Show/Hide -> Show Message Reuses (рис.56). У результаті часовий графік буде представлено у наступному виді, як показано на рис.57.

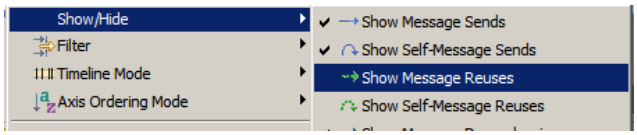

Рис. 56 – Меню включення відображення повторного пересилання.

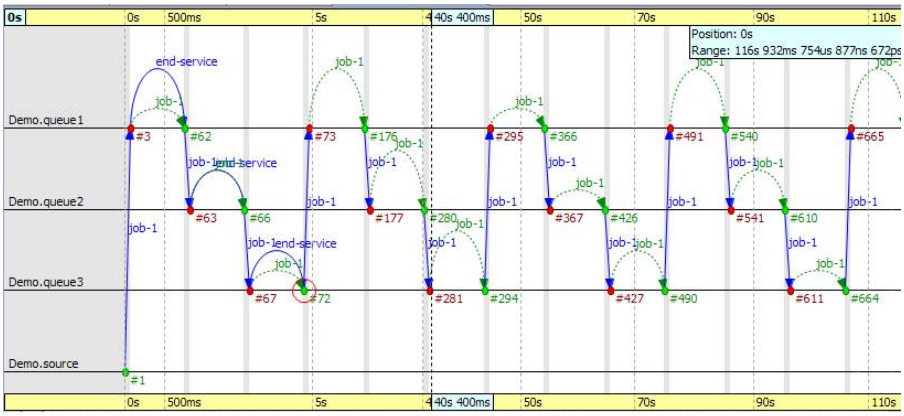

Рис. 57 – Часовий графік

3.7 Висновки.

Було розроблено перший проект моделі, побудована перша примітивна мережа засобами візуального розташування об’єктів на мережі та шляхом написання програмного коду мовою NED.

Запуск першого проекту мережі надав результати моделювання, було проведено аналіз результатів для різних апріорних даних.

Було показано механізми налаштування апріорних даних проекту та засоби аналізу результатів. Було показано та проведено аналіз графіків черг, шляхів пакетів та фільтрації пакетів у часі.

Контрольні питання

1. Приклад створення нового проекту.
2. Приклад створення NED файлу
3. Які типи підмодулів доступні з палітри компонентів?
4 Принципи конфігурування моделі. Структура конфігураційного файлу.
5. Логування процесу імітації. Типи лог-файлів.
6. Конфігурування запуску моделі. Які середовища запуску моделей використовує Omnet++?
7. Які файли створюються після завершення сеансу імітації в папці проекту?
8. Аналіз результатів моделювання. Типи файлів аналізів.
9. Яку інформацію надає графік послідовностей подій?
10 Яку інформацію надає часовий графік з нелінійним часом?

4. МОВА ОПИСУ МЕРЕЖІ NED

Структура імітаційної моделі описується засобами мови NED, яка означає Network Description. NED дозволяє користувачеві поєднати прості модулі, а також з'єднати й зібрати їх у складені модулі. Користувач може позначити деякі складені модулі, як мережі; тобто автономні імітаційні моделі. Канали – інший тип компонентів, екземпляри якого можуть бути також використані в складених модулях. Мова NED має ряд особливостей, які дозволяють їй добре масштабуватися для великих проектів: ієрархічність, підтримка компонентом структури, робота з інтерфейсами, об'єднання в пакети, внутрішні типи, анотація метаданих.

Мова NED має деревоподібну структуру, код може бути перетворений в XML і назад без втрати даних, включаючи коментарі.

4.1 Базові позначення

Припустимо, що дана деяка мережа зв'язку, яка складається з вузлів. На кожному вузлі працює додаток, який генерує пакети через випадкові проміжки часу. У свою чергу вузли самі є маршрутизаторами. Передбачається, що додатки використовують зв'язок на основі дейтаграм, що дозволяє не враховувати транспортний рівень у моделі.

Розглянемо приклад мережі, топологія якої представлена на рис.58. Відповідний їй код мови NED буде виглядати таким чином:

```
// network description
network Network
{
   submodules:
      node1: Node;
      node2: Node;
      node3: Node;
      ...
   connections:
      node1.port++ <--> {datarate=100Mbps;} <--> node2.port++;
      node2.port++ <--> {datarate=100Mbps;} <--> node4.port++;
      node4.port++ <--> {datarate=100Mbps;} <--> node6.port++;
...
}
```

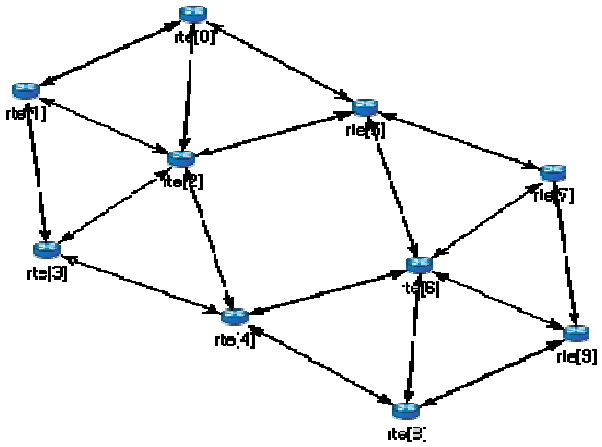

Рис.58 – Топологічна структура мережі

Отриманий код описує тип мережі за назвою Network. У розділі submodules показано, що дана мережа містить вузли з іменами node1, node2 і т.д., які мають тип NED модуля Node. У свою чергу тип Node буде визначено далі.

Розділ декларації connections визначає характеристики з'єднання вузлів. Подвійна стрілка означає двунаправлене з'єднання. Точки підключення модулів називаються ворота (Gates), а позначення port++ додає нові ворота (порт) до вектора (колекції), який містить список воріт вузла. Вузли з'єднані з каналом, який має швидкість передачі даних 100 Мбіт.

Наведений вище код міститься у файл із іменем Net6.ned. Існує угода розміщати кожний NED код в окремому файлі, хоча не обов'язково робити так. Можна визначати будь-яку кількість мереж в NED файлі, тоді для кожного сеансу моделювання необхідно вказувати, яку мережу використовувати в моделі. Стандартним способом указують необхідну мережу через параметр network у конфігураційному файлі (за замовчуванням файл omnetpp.ini):

```
[General]
network = Network
```

Досить громіздко повторювати запис швидкості передачі даних для кожного з'єднання в розділі connections. Засобами мови NED можна створити новий тип – канал (channel), який інкапсулює налаштування швидкості передачі даних. Тип *канал* може бути визначений усередині мережі, так що він не буде засмічувати глобальний простір імен (global namespace).

Модифікований запис коду опису мережі Network буде виглядати в таким чином:

```
// A Network
//
network Network
{
    types:
        channel Ch extends ned.Dataratechannel {
            datarate = 100Mbps;
        }
    submodules:
        node1: Node;
        node2: Node;
        node3: Node;
        ...
    connections:
        node1.port++ <--> Ch <--> node2.port++;
        node2.port++ <--> Ch <--> node4.port++;
        node4.port++ <--> Ch <--> node6.port++;
...
}
```

Прості модулі (Simple modules) є основними будівельними блоками для інших складених модулів. Уся активна поведінка моделі інкапсулюється в простих (Simple) модулях. Поведінка визначається за допомогою класів C ++; NED файли тільки описують зовні видимий інтерфейс модуля (ворота, параметри).

У розглянутому випадку можна було б визначити кожний вузол Node у якості простого модуля. Проте, його функціональність досить складна (генерація трафіка, маршрутизація і т.д.), так що краще реалізувати його з декількох більш дрібних типів простих модулів, які зберуться в складений модуль. Таким чином, необхідно створити один простий модуль для генерації трафіка – App, один для маршрутизації – Routing, і один для керування чергою пакетів, які відправляються, – Queue. Для стислості викладу поки не будемо розглядати код останніх двох модулів.

```
simple App
{
    parameters:
        int destaddress;
        ...
        @display("i=block/browser");
```

```
    gates:
        input in;
        output out;
}

simple Routing
{
    ...
}

simple Queue
{
    ...
}
```

За згодою, імена типів модулів починаються з великої букви, а імена параметрів і воріт починаються з нижнього регістру. Мова NED чутлива до регістру. Таким чином, коди простих модулів будуть розміщені у файлах App.ned, Routing.ned і Queue.ned.

Розглянемо програмний код простого модуля App, який має параметр Destaddress, а також двоє воріт для відправлення й приймання пакетів додатків – output і input відповідно.

Аргумент @display() називають рядком відображення, він визначає візуалізацію модуля в графічних середовищах; "i=..." – визначає іконку для модуля за замовчуванням.

Як правило, ключові слова, які починаються із символу @, наприклад, @display, називаються в мові NED властивостями й використовуються для анотування різних об'єктів з метаданими. Властивості можуть бути прикріплені до файлів, модулів, параметрів, воріт, з'єднань і іншим об'єктам. Також значення параметрів мають дуже гнучкий синтаксис.Тепер можна зібрати модулі генерації трафіка App, маршрутизації Routing і керування чергами Queue у якості елементів складеного (Compound) модуля Node, рис.59. Складений модуль групує в собі інші підмодулі й надалі, у свою чергу, може бути також використаний у якості будівельного блоку для інших модулів. Мережі також є свого роду складеними модулями.

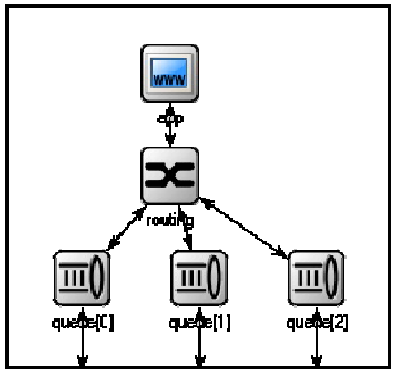

Рис. 59 – Структура складеного модуля Node

```
module Node
{
    parameters:
        int address;
        @display("i=misc/node_vs,gold");
    gates:
        inout port[];
    submodules:
        app: App;
        routing: Routing;
        queue[sizeof(port)]: Queue;
    connections:
        routing.localout --> app.in;
        routing.localin <-- app.out;
        for i=0..sizeof(port)-1 {
            routing.out[i] --> queue[i].in;
            routing.in[i] <-- queue[i].out;
            queue[i].line <--> port[i];
    }
}
```

Складені модулі, як і прості модулі, можуть мати параметри й ворота. Модуль Node містить параметр address, вектор невизначеного розміру port, що містить список воріт. Ворота мають тип даних inout, що описує двунаправлену зв'язок.

Підмодулі, що входять до складу модуля, перераховані в розділі submodules. Складений модуль Node містить підмодулі генерації трафіка app, маршрутизації routing і вектор підмодулів queue[ ], який містить один модуль типу Queue для кожного порту.

У розділі connections описуються з'єднання підмодулів один з одним і з батьківським модулем. Одиночні стрілки описують з'єднання вхідних (in) і вихідних (out) воріт, подвійні стрілки описують з'єднання двунаправлених воріт, типу inout. Цикл for описує з'єднання модуля routing з кожним модулем вектора queue і з'єднання вхідного й вихідного з'єднання (line) воріт черг queue з відповідним йому портом port складеного модуля Node.

Створені описи типів мовою NED при запуску моделювання завантажуються з NED файлів. Програма повинна вже містити C ++ класи, які реалізують необхідні прості модулі: App, Routing і Queue. Їх C ++ код є або частиною файлу, що виконується, або завантажується із загальної бібліотеки. Програма моделювання також завантажує конфігурацію з файлу omnetpp.ini, у якій визначено, що імітаційною моделлю, яку необхідно запустити буде мережа Network. Після чого мережа готова для моделювання.

Побудова імітаційної моделі виконується зверху вниз, починаючи з порожнього системного модуля, потім створюються всі підмодулі, їхні параметри, призначаються розмірності векторів воріт, повністю виконується їхнє з'єднання, потім будуються внутрішності підмодулів.

4.2 Прості модулі

Прості модулі (Simple modules) є активними компонентами моделі. Прості модулі визначаються за допомогою ключового слова simple.

Приклад простого модуля, який описує чергу:

```
simple Queue
{
    parameters:
        int capacity;
        @display("i=block/queue");
    gates:
        input in;
        output out;
}
```

Розділи parameters і gates є необов'язковими, і можуть бути відсутні, якщо в модуля немає параметрів або воріт. Крім того, і ключове слово parameters є необов'язковим, його можна не писати, навіть якщо є параметри або властивості.

Визначення модуля мовою NED не містить ніякого коду, що описує внутрішнє функціонування модуля. Функціонування модулів написано мовою C ++, і за замовчуванням, OMNeT++ шукає класи C++ з тим же іменем, що в описаного типу NED модуля (у цьому випадку Queue).

Можна явно вказати C++ клас у властивості @class. Можливий варіант запису класу із вказівкою простору імен, як показано в наступному прикладі, який використовує клас черг Mylib::Queue з бібліотеки Mylib.

```
simple Queue
{
   parameters:
      int capacity;
      @class(mylib::Queue);
      @display("i=block/queue");
   gates:
      input in;
      output out;
}
```

Якщо використовується кілька модулів, які перебувають у загальному просторі імен (в одній бібліотеці), то замість властивості @class можна використовувати інструкцію @namespace. Простір імен C++, зазначене в @namespace, буде додано до імен класів. У наступному прикладі використовуються C++ класи Mylib::App, Mylib::Router і mylib::Queue.

```
@namespace(mylib);
simple App {
   ...
}
simple Router {
...
}
simplequeue {
...}
```

Інструкція @namespace може бути задана не тільки на рівні файлів, при розміщенні її у файлі package.ned, простір імен буде застосовуватися до всіх файлів і папок у тому ж каталозі.

Класи реалізації модулів на C++ повинні бути підкласами класу csimplemodule.

Прості модулі можуть бути розширені за допомогою підкласів з метою встановити фіксовані значення деяким відкритим параметрам або розмірам воріт, або замінити C++ клас із інших.

У прикладі показано, як спеціалізувати модуль, встановивши параметру фіксоване значення не змінюючи C++ клас.

```
simple Queue{
  int capacity;
  ...
}

simple Boundedqueue extends Queue{
    capacity = 10;
}
```

Допустимо, є написаний на C++ клас пріоритетної черги Priorityqueue, необхідно створити відповідний NED тип, успадкований від Queue. Запис у прикладі нижче буде працювати не так, як очікується.

```
simple Priorityqueue extends Queue{   // wrong! still uses the Queue C++ class
}
```

Правильним записом буде додавання властивості @class, щоб перевизначити спадкування класу C ++.

```
simple Priorityqueue extends Queue {
  @class(Priorityqueue);
}
```

4.3 Складені модулі

Складений модуль групує інші модулі у великий агрегат. Складений модуль може мати ворота (Gates) і параметри, як простий модуль, але ніякої активної поведінки він не виконує.

Коли необхідно додати код C++ для складеного модуля, краще інкапсулювати цей код у простий модуль, і додати його в якості підмодуля в складений.

Оголошення складеного модуля містить кілька розділів, кожний з яких не є обов'язковим:

```
module Host
{
  types:
     ...
  parameters:
     ...
  gates:
...
submodules:
     ...
connections:
     ...
}
```

Модулі, що містяться усередині складеного модуля, називаються підмодулями, вони перераховані в секції submodules. Можна створювати масиви підмодулів (тобто векторів підмодулів). З'єднання перераховані в розділі connections. Можна задавати з'єднання, використовуючи прості конструкції програмування (цикл, умовне вираження). Поведінка з'єднання визначається шляхом зв'язування каналу із з'єднанням. Тип каналу залежить від параметра.

Типи модулів і каналів, використовуваних тільки локально, визначають у розділі types у вигляді внутрішніх типів, так що вони не забруднюють простір імен.

Складені модулі можна розширювати шляхом створення підкласів. Через механізм спадкування (Inheritance) можна додавати нові підмодулі, нові зв'язки, параметри й ворота. Крім того, можна посилатися на успадковані підмодулі, успадковані типи і т.д.

У наступному прикладі показано, як зібрати в групу загальні протоколи для бездротових хостів, а користувацькі (User Agent) протоколи додавати через механізм підкласів.

```
module Wirelesshostbase
{
  gates:
     input radioin;
  submodules:
     tcp: TCP;
     ip: IP;
```

```
        wlan: Ieee80211;

    connections:
        tcp.ipout --> ip.tcpin;
        tcp.ipin <-- ip.tcpout;
        ip.nicout++ --> wlan.ipin;
        ip.nicin++ <-- wlan.ipout;
        wlan.radioin <-- radioin;
}

module Wirelesshost extends Wirelesshostbase
{
  submodules:
      webagent: Webagent;
  connections:
      webagent.tcpout --> tcp.appin++;
      webagent.tcpin<-- tcp.appout++;
}
```

Складений модуль Wirelesshost може бути додатково розширений, наприклад, додаванням порту Ethernet:

```
module Desktophost extends Wirelesshost
{
  gates:
      inout ethg;
  submodules:
      eth: Ethernetnic;
  connections:
      ip.nicout++ --> eth.ipin;
      ip.nicin++ <-- eth.ipout;
      eth.phy <--> ethg;
}
```

4.4 Канали

Канали інкапсулюють параметри й поведінку з'єднань вузлів. Канали подібно простим модулям складаються із класів C++. Правила знаходження

класу C ++ для відповідного канального типу такі ж, як у простих модулів: ім'я класу за замовчуванням збігається з іменем NED типу, якщо не зазначена властивість @class або інструкція @namespace, а клас C++ успадковується при створенні підкласу каналу.

Таким чином тип каналу, що випливає, буде мати на увазі існування C ++ класу Customchannel:

```
channel Customchannel  // requires a Customchannel C++ class
{
```

Практична різниця в порівнянні з модулями полягає в тому, що досить рідко необхідно створювати власний C++ клас каналу, тому що існують визначені типи каналів, на основі яких можна створити підклас, успадковуючи їх C++ код. Пропонуються наступні визначені типи каналів: ned.Idealchannel, ned.Delaychannel і ned.Dataratechannel (з пакета «Ned»).

Тип Idealchannel не має параметрів і дозволяє передавати повідомлення без затримки або будь-якого побічного ефекту. З'єднання вузлів без об'єкта каналу й з'єднання через Idealchannel поводяться однаково. Проте, Idealchannel має застосування, наприклад, коли об'єкт каналу потрібен для того, щоб він міг містити в собі нові властивості або параметри, які будуть прочитані іншими частинами імітаційної моделі.

Тип Delaychannel має два параметри:

• delay – має тип даних подвійної точності (double), являє собою затримку поширення повідомлення. Числові значення вказуються разом з одиницею часу (сек, мсек, і т.д.).

• disabled – має логічний тип даних, значення за замовчуванням false; якщо встановлене значення true, те об'єкт каналу буде відфільтровувати усі повідомлення.

Тип Dataratechannel має кілька додаткових параметрів у порівнянні з Delaychannel:

• Datarate – має тип даних подвійної точності (double), являє собою швидкість передачі даних каналу. Значення за замовчуванням вказуються в бітах у секунду або додаються одиниці виміру (bps, kbps, Mbps, Gbps). За замовчуванням параметру задається нульове значення, яке обробляється особливим образом і відповідає нульовій тривалості передачі, що характеризує нескінченну смугу пропущення. Швидкість передачі даних використовується для обчислення тривалості передачі пакетів.

• параметри ber і per відповідають частоті появи помилкових битів (Bit Error Rate) і частоті появи помилкових пакетів (Packet Error Rate). Ці параметри

відповідають за основні функції моделювання помилок. Параметри приймають значення подвійної точності (double) у діапазоні [0,1]. Коли модуль каналу приймає рішення (на підставі випадкових чисел), що повинна відбутися помилка під час передачі пакета, він встановлює прапор помилки в об'єкті пакета. Модуль приймача завжди перевіряє прапор помилки пакета й, якщо він встановлений, відкидає цей пакет як пошкоджений. За замовчуванням параметри ber і per дорівнюють нулю.

У наступному прикладі показано, як створити новий тип каналу за рахунок спеціалізації Dataratechannel:

```
channel Ethernet100 extends ned.Dataratechannel
{
   datarate = 100Mbps;
   delay = 100us;
   ber = 1e-10;
}
```

Можна додавати нові параметри й властивості класам каналів, використовуючи механізм підкласів, або можна змінювати існуючі. У наступному прикладі показано, як задати затримку поширення, що залежить від відстані (довжини каналу). Параметр delay обчислюється з використанням відповідного співвідношення.

```
channeldataratechannel2 extendsned.Dataratechannel
{
   double distance @unit(m);
   delay = this.distance / 200000km * 1s;
}
```

Параметри модуля в основному використовуються в якості вхідних даних для C ++ класу, що міститься в його основі, але навіть, якщо повторно використовувати C ++ класи вбудованих типів каналів, вони можуть бути прочитані й використані іншими частинами моделі. Наприклад, доданий параметр cost (або властивість @cost), відповідальний за вартість передачі, може використовуватися алгоритмами маршрутизації для прийняття рішень. У наступному прикладі доданий параметр cost і властивість @backbone, що вказує, що використовується канал опорної мережі Інтернет.

```
channel Backbone extends ned.Dataratechannel
{
  @backbone;
  double cost = default(1);
}
```

4.5 Параметри

Параметри (Parameters) є змінними, які належать модулю. Параметри можуть використовуватися при побудові топології (кількість вузлів і т.д.), а також як вхідні параметри C ++ коду, який реалізує прості модулі й канали.

Параметри бувають наступних типів: double, int, bool, string і XML. вони також можуть бути оголошені volatile (перерозрахунковий). Для числових типів, можна вказувати одиницю виміру (у властивості @unit), щоб підвищити узгодженість типу.

Параметри можуть одержати свої значення з NED файлів або з конфігурації (omnetpp.ini). Значення за замовчуванням задається директивою default (...), і використовується, коли значення параметру не було привласнено в майбутньому.

Наступний приклад демонструє опис простого модуля, який має п'ять параметрів, три з яких мають значення за замовчуванням:

```
simple App
{
   parameters:
      string protocol;       // protocol to use: "UDP" / "IP" / "ICMP" / ...
      int destaddress;       // destination address
      volatile double sendinterval @unit(s) = default(exponential(1s));
                        // time between generating packets
      volatile int packetlength @unit(byte) = default(100B);
                        // length of one packet
      volatile int timetolive = default(32);
                        // maximum number of network hops to survive
   gates:
      input in;
      output out;
}
```

Задавати значення параметрам можна декількома способами: з NED коду, з конфігурації (omnetpp.ini) або в інтерактивному режимі їх може ввести користувач. Мова NED дозволяє задавати значення параметрам у декількох місцях програми: у підкласах через спадкування; у підмодулях і визначеннях з'єднань (connections), де інстанцується (створюється екземпляр класу) тип

NED; у мережах і складених модулях, які безпосередньо або побічно містять відповідний підмодуль або з'єднання.

Наприклад, можна було б спеціалізувати описаний вище тип модуля App через спадкування в таким чином:

```
simple Pingapp extends App
{
   parameters:
      protocol = "ICMP/ECHO"
      sendinterval = default(1s);
      packetlength = default(64byte);
}
```

Цей код встановлює параметру протоколу фіксоване значення "ICMP/ECHO" і змінює значення, задані за замовчуванням, параметрам sendinterval і packetlength. Значення парметра protocol із цього моменту заблоковано на рівні опису класу Pingapp, воно не може бути змінено за допомогою створення підкласів або будь-якими іншими способами. Параметри sendinterval і packetlength дотепер невизначені в цім місці, але їх значення за замовчуванням були змінені.

Тепер, розглянемо, визначення складеного модуля Host, який у своєму составі використовує Pingapp у якості підмодуля:

```
module Host
{
   submodules:
      ping : Pingapp {
         packetlength = 128B; // always ping with 128-byte packets
}
      ...
}
```

У цім визначенні параметру packetlength встановлюється фіксоване значення. Зараз жорстко записано (hardcoded), що Host відправляє 128 байтні пінг-пакети; дане значення не може бути змінено надалі в коді NED або через конфігурацію.

Можна задавати значення параметрам не тільки зі складеного модуля, який містить даний підмодуль, а також і з модулів, що перебувають вище в дереві

модулів. Якщо є мережа, що містить кілька модулів Host, вона визначається таким чином:

```
network Network
{
   submodules:
      host[100]: Host {
         ping.timetolive = default(3);
         ping.destaddress = default(0);
}
      ...
}
```

Установка значень параметрів також може бути поміщена в параметри блоку модуля вихідного з'єднання, що забезпечує додаткову гнучкість. Наступне визначення задає в рамках мережі вектору параметрів host значення destaddress таким чином, що половина елементів вектора пінгує адресу 50, а інша половина пингує адресу 0:

```
network Network
{
   parameters:
      host[*].ping.timetolive = default(3);
      host[0..49].ping.destaddress = default(50);
      host[50..].ping.destaddress = default(0);
   submodules:
host[100]: Host;
      ...
}
```

Використання зірочки в команді host[*] відповідає будь-якому індексу з можливого діапазону. Якщо було задано декілька окремих пронумерованих змінних host замість вектора підмодулів, то визначення мережі буде виглядати таким чином:

```
network Network{
   parameters:
      host*.ping.timetolive = default(3);
      host{0..49}.ping.destaddress = default(50);
```

```
        host{50..}.ping.destaddress = default(0);
    submodules:
        host0: Host;
        host1: Host;
        ...
       host99: Host;
}
```

Тут символ «зірочка» (*) відповідає будь-якому підрядку, що не містить крапку, а «дві крапки» (..) усередині пари фігурних дужок відповідає натуральному числу, вставленому в рядок.

У деяких визначеннях параметрів, що зустрічалися вище, з лівої сторони від знака рівняння містилась крапка й, найчастіше, підстановка (wildcard) - зірочка або числовий діапазон; цей спосіб завдання імен параметрів називається завданням по шаблонові (pattern assignments).

Ще один груповий символ, який може бути використаний у шаблонах – «дві зірочки» (**); він задає будь-яку послідовність символів, включаючи крапки, таким чином, він може відповідати декільком елементам шляху.

Наприклад:

```
network Network
{
    parameters:
        **.timetolive = default(3);
        **.destaddress = default(0);
    submodules:
        host0: Host;
        host1: Host;
        ...
}
```

Тут деякі оголошення параметрів у наведенних вище прикладах змінюють значення за замовчуванням, у той час як інші оголошення параметрів змінюють фіксовані значення. Параметри, які не одержали ніякого фіксованого значення в NED файлах можуть бути задані в конфігурації (omnetpp.ini).

У свою чергу, значення не за замовчуванням, призначені з NED, не можуть бути перезаписані пізніше в коді NED або в ini-файлах, вони стають жорстко зафіксованими (hardcoded). Параметр може бути призначений у файлі конфігурації, використовуючи аналогічний синтаксис запису шаблону NED:

```
Network.host[*].ping.sendinterval = 500ms  # for the host[100] example
Network.host*.ping.sendinterval = 500ms    # for the host0,host1,... example
**.sendinterval = 500ms
```

Часто використовують подвійну зірочку, щоб менше друкувати. Можна задати аналогічний запис таким чином:

```
**.ping.sendinterval = 500ms
```

Якщо відомо, що в програмі немає якого-небудь іншого параметра setinterval, то можна написати

```
**.sendinterval = 500ms
```

Можна також записувати вираз, у тому числі стохастичні вирази, як у файлах NED, так і в ini файлах. Наприклад, таким чином можна задати джиттер (jitter) для відправлення пінг-пакетів:

```
**.sendinterval = 1s + normal(0s, 0.001s)  # or just: normal(1s, 0.001s)
```

Якщо немає явного присвоювання для параметра в NED, або в ini файлі, то буде застосовуватися неявно значення за замовчуванням =default(...). Якщо значення за замовчуванням не задано, тоді користувачеві буде запропоновано ввести значення самому, за умови, що програма моделювання дозволяє це зробити; а якщо ні, то буде видано помилку (інтерактивний режим звичайно відключений).

Крім того, можна явно застосувати значення за замовчуванням:

```
**.sendinterval = default
```

І, нарешті, можна явно вказати моделі, щоб вона запитувала в користувача значення в інтерактивному режимі (за умови, що інтерактивність включено):

```
**.sendinterval = ask
```

При виборі того, чи задавати значення параметру в NED коді або в ini файлі, необхідно знати, що перевага INI файлів полягає в тому, що вони дозволяють більш якісно відокремлювати модель від експериментів над нею. З одного боку, NED файли (разом з кодом C ++) вважаються частиною моделі й більш-менш постійні. З іншого боку, INI файли призначені для експериментів з моделлю, дозволяючи запускати її кілька раз із різними наборами значень параметрів. Таким чином, параметри, які будуть змінені залежно від експерименту, повинні бути введені в INI файлі.

Значення параметрів можуть бути дані за допомогою виразів. Вирази мови NED мають C++ подібний синтаксис, з деякими варіаціями щодо імен операторів.

Вирази можуть посилатися на параметри (parameters) модуля, вектори воріт (gates), розмір вектора модулів (оператор sizeof) або індекс поточного модуля у векторі підмодулів (index).

Вирази можуть посилатися на параметри складеного модуля, звітного (із префіксом this.) модуля, а також на параметри вже певних підмодулів, використовуючи синтаксиси submodule.parametername або submodule [index].parametername.

Модифікатор параметра volatile (перерахунковий) вказує на те, що заданий вираз для обчислення значення параметра викликається щораз, коли зчитується значення параметра. Модифікатор volatile вказується параметру, якщо його вираз (формула) не є постійною величиною, наприклад, воно містить у собі номера, взяті з генератора випадкових чисел. А якщо ні, то, параметри без модифікатора volatile обчислюються тільки один раз, тобто вирази обчислюються й заміняються результуючою константою на самому початку моделювання.

Припустимо, є простий модуль Queue, у якого заданий перерозрахунковий параметр типу double з іменем servicetime.

```
simple Queue
{
   parameters:
      volatile double servicetime;
}
```

Через те що встановлений модифікатор volatile, програмна реалізація модуля черги на C ++ буде зчитувати параметр servicetime щораз, коли буде необхідно його значення; тобто для кожного завдання, що обслуговується (job). Таким чином, параметру servicetime буде привласнюватись вираз на подобі uniform(0.5s, 1.5s), причому для кожної роботи буде отримано різний випадковий час обслуговування. У наступному прикладі показано, як можна одержувати параметр мінливий у часі за рахунок використання NED функції simtime(), яка повертає поточний час моделювання:

```
**.servicetime = simtime()<1000s ? 1s : 2s  # queue that slows down after 1000s
```

На практиці, модифікатор параметра volatile, як правило, використовуються для налаштування джерела випадкових чисел у модулях.

Можна крім числового значення під час оголошення параметра задавати пов'язану з ним одиницю виміру, додавши властивість @unit. Наприклад:

```
simple App
{
```

```
    parameters:
        volatile double sendinterval @unit(s) = default(exponential(350ms));
        volatile int packetlength @unit(byte) = default(4Kib);
...
}
```

Модифікатори @unit(s) і @unit(byte) задають одиницю виміру для параметра. Значення, привласнені параметрам, повинні мати однакову або сумісну одиницю виміру. Таким чином, змінна з модифікатором @unit(s) може отримувати значення, задані в мілісекундах, наносекундах, хвилинах, годинах і т.д., а з модифікатором @unit (байт) - у кілобайтах, мегабайтах і т.д.

В OMNeT++ виконується повний і строгий контроль одиниць вимірів у параметрів, щоб забезпечити погодженість їх використання. У свою чергу, константи завжди повинні включати одиниці виміру. Властивість @unit параметрів не можна додавати або перевизначати в підкласах або в описах підмодулів.Іноді модулям необхідні складні структури вхідних даних, які не можуть бути задані параметрами модуля. Для розв'язку проблеми вхідні дані розміщають у користувацькому файлі конфігурації, а ім'я файлу передають модулю в якості параметра рядка, у свою чергу, модуль читає й розбирає структуру файлу.

Завдання спрощується, якщо конфігурація використовує синтаксис XML, оскільки OMNeT++ містить вбудовану підтримку XML файлів. За допомогою синтаксичного аналізатора XML (бібліотеки Libxml2 і Expat), OMNeT++ читає й перевіряє тип документа (Dtd-validation) файлу (якщо документ XML містить директиву DOCTYPE), кэшує файл (не дивлячись на багаторазове посилання на нього з декількох модулів, завантаження виконується один раз), дозволяє вибрати потрібну частину документа, використовуючи мову Xpath, і представляє вміст у вигляді дерева об'єктів за принципом об'єктної моделі документа (DOM).

Ця можливість задається через NED параметр типу xml або функцію xmldoc(). Можна вказати Xml-тип параметру модуля, задавши конкретний файл XML (або додатково вказати елемент усередині файлу XML) за допомогою функції xmldoc(). Можна вказувати параметру тип XML як у мові NED, так і в конфігурації omnetpp.ini.

У прикладі оголошується параметр типу xml, і йому привласнюється файл XML. Ім'я файлу визначається щодо робочої директорії.

```
simple Trafgen {
    parameters:
        xml profile;
```

```
    gates:
        output out;
}
module Node {
    submodules:
        trafgen1 : Trafgen {
            profile = xmldoc("data.xml");
        }
        ...
}
```

Крім того, можна привласнити параметру модуля значення окремого елемента в складі XML файлу. Це корисно, якщо необхідно згрупувати вхідні дані декількох модулів в один загальний XML файл. Наприклад, що випливає XML файл містить два профілі з ідентифікаторами (id) gen1 і gen2:

```
<?xml>
<root>
    <profile id="gen1">
        <param>3</param>
        <param>5</param>
    </profile>
        <profile id="gen2">
        <param>9</param>
</profile>
</root>
```

І можна привласнити кожному окремому підмодулю відповідний йому профіль, використовуючи вирази мови Xpath:

```
module Node {
    submodules:
        trafgen1 : Trafgen {
            profile = xmldoc("all.xml", "/root/profile[@id='gen1']");
        }
        trafgen2 : Trafgen {
            profile = xmldoc("all.xml", "/root/profile[@id='gen2']");
        }
}
```

Крім того, можна створити документ XML через строкову константу, використовуючи функцію xml(), що корисно для створення значення за замовчуванням для параметрів з типом xml. Наприклад:

```
simple Trafgen {
   parameters:
      xml profile = xml("<root/>"); // empty document as default
...
}
```

Функція xml() аналогічна xmldoc(), але також дозволяє задавати необов'язковий другий параметр Xpath для вибору піддерева шляху.

4.6 Ворота (Gates)

Ворота являють собою точки підключення модулів. OMNeT++ пропонує три типи воріт: вхід (in), вихід (out) і двунаправленні (inout), причому останній тип – це, по суті, з'єднаний одночасно вхід і вихід воріт.

Ворота, будь то вхід або вихід, можуть бути підключені тільки до одних інших воріт. Для складених модулів ворота означають одне з'єднання для вихідних потоків за межі модуля, а інше для вхідних з'єднань до середини.

Можна створювати окремі ворота й вектори воріт. Розмір вектора воріт задається у квадратних дужках під час оголошення, але також можна, залишати довжину не заданою, указавши пари порожніх дужок "[ ]".

Якщо розмір вектора воріт залишається незаданим, то його можна визначити пізніше, під час спадкування модуля або при використанні в якості підмодуля складеного модуля. Проте, значення однаково може бути не задано, тому що можна створювати нові з'єднання оператором gate++, який автоматично збільшує розмір вектора воріт.

Розмір вектора воріт може бути отриманий з різних місць NED коду оператором Sizeof().

В загалі мова NED вимагає, щоб усі ворота були підключені. Щоб зм'якшити ця вимогу, можна вказати воротам властивість @loose, яка відключає перевірку підключення для заданих воріт. Крім того, вхідні ворота, які використовуються для одержання повідомлень через функцію senddirect() повинні мати параметр @directin. Крім того, можна відключити перевірку

підключення всім воротам всередині складеного модуля, вказавши ключове слово allowunconnected у розділі connections модуля.

У наступному прикладі, модуль Classifier має один вхід для приймання завдань, які він буде посилати на один з виходів. Число виходів визначається параметром модуля:

```
simple Classifier {
  parameters:
     int numcategories;
  gates:
      input in;
      output out[numcategories];
}
```

Наступний модуль Sink також має незадану розмірність вектора вхідних воріт in[ ], таким чином, він може бути з'єднаний з різними модулями:

```
simple Sink
{
   gates:
      input in[];
}
```

Наступний код визначає вузол, використовуваний для побудови квадратної сітки. Очевидно, що вузли по краях сітки містять деякі ворота, які ні із чим не зв'язані. Ця можливість виходить шляхом додаванням анотації @loose:

```
simple Gridnode
{
   gates:
      inout neighbour[4] @loose;
}
```

Бездротовий вузол Wirelessnode, одержує повідомлення шляхом прямої радіопередачі, тому його ворота radioin відзначені анотацією @directin.

```
simple Wirelessnode
{
   gates:
```

```
        input radioin @directin;
}
```

У наступному прикладі визначений модуль Treenode, використовуваний для деревоподібної структури. Він може підключати будь-яку кількість нащадків (children). Потім, модуль Binarytreenode успадковує його й перевизначає розмір вектора воріт рівним двом:

```
simple Treenode {
    gates:
        inout parent;
        inout children[];
}
simple Binarytreenode extends Treenode
{
    gates:
         children[2];
}
```

У наступному прикладі розмір вектора воріт установлюється в підмодулі, використовуючи той же тип модуля Treenode:

```
module Binarytree
{
    submodules:
          nodes[31]: Treenode
          {
              gates:
                  children[2];
          }
    connections:
        ...
}
```

4.7 Підмодулі

Модулі, з яких полягає складений модуль, називаються його підмодулями. Підмодуль має ім'я і є екземпляром складеного або простого модуля. В NED

визначенні підмодуля, тип модуля звичайно задається статично, але можна також указати тип строковим виразом.

Мова NED підтримує масиви (вектори) підмодулів і умовні підмодулі. Розмір вектора підмодулів, на відміну від розміру вектор воріт, завжди повинен бути зазначений і не може бути залишений невизначеним, як у випадку з воротами.

Можна додавати нові підмодулі до існуючого складеного модуля через механізм спадкування. Базовий синтаксис опису підмодуля наведений нижче:

```
module Node
{
   submodules:
      routing: Routing;   // a submodule
      queue[sizeof(port)]: Queue;  // submodule vector
...
}
```

Як було показано в попередніх прикладах коду, тіло підмодуля задається у фігурних дужках, де можна задати параметри, установити розмір векторів воріт, і додавати або змінювати властивості, такі як рядок візуалізації @display. З іншого боку, не можна додавати нові параметри й ворота.

Рядки візуалізації, зазначені тут, будуть об'єднані з рядками візуалізації в описі типу, використовуючи алгоритм злиття.

```
modulenode
{
gates:
      inoutport[ ];
submodules:
      routing: Routing {
         parameters:   // this keyword is optional
            routingtable = "routingtable.txt"; // assign parameter
         gates:
            in[sizeof(port)];  // set gate vector size
            out[sizeof(port)];
      }
      queue[sizeof(port)]: Queue {
         @display("t=queue id $id"); // modify display string
         id = 1000+index;  // use submodule index to generate different Ids
```

```
}
connections:
    ...
}
```

Порожнє тіло модуля може бути опущене, тобто, запис queue: Queue; буде відповідати запису

```
queue: Queue {
 }
```

Підмодуль або вектор підмодулів можуть бути умовними. Після типу модуля задають умови використання цього типу. Вони задаються ключовим словом if із вказівкою умови, як у наведеному нижче прикладі:

```
module Host
{
   parameters:
      bool withtcp = default(true);
   submodules:
      tcp : TCP if withtcp;
...
}
```

Умови з векторами підмодулів використовуються досить рідко.

4.8 З'єднання (Connections)

З'єднання визначаються в розділі connections складових модулів. З'єднання не поширюються на різні рівні ієрархії; можна з'єднати двоє воріт підмодуля, ворота підмодуля й вхідні ворота батьківського складеного модуля, або двоє воріт батьківського модуля між собою, але неможливо підключити з'єднання до будь-яких воріт зовні батьківського модуля, або усередині складових підмодулів.

З'єднання вхідних (in) і вихідних (out) воріт описуються звичайною стрілкою, а двунаправлених (inout) воріт подвійною стрілкою "<-->". Для з'єднання двох воріт каналом зв'язку використовують дві стрілки, між якими вказують специфікацію каналу. Той же синтаксис використовується для додавання властивостей з'єднанню, таких як @display.

Ворота визначаються записом modulespec.gatespec (для підключення воріт підмодуля), або gatespec (для підключення воріт складеного модуля). Запис modulespec відповідає імені підмодуля (для скалярних підмодулів), або потрібно додавати до імені підмодуля індекс у квадратних дужках (для векторів підмодулів). Для скалярних воріт ім'я записується як gatespec; для векторів воріт до імені додається індекс у квадратних дужках, або запис робиться через команду gatename++.

Нотація gatename++ позначає, що викликається перший невикористаний індекс воріт, який буде використовуватися надалі. Якщо всі ворота даного вектора воріт вже використовуються, то виконання команди gatename++ для підмодулів, приведе до того, що вектор воріт буде розширений на одну одиницю, а для складених модулів виконання команди gatename++ при всіх зайнятих воротах видасть помилку.

Коли оператор ++ використовується із суфіксом $i або $o (наприклад. g$i++ або g$o++), це значить, що буде додано пару воріт вхід-вихід (input+output) для підтримки рівного розміру воріт в обох напрямкахХарактеристики каналу (->channelspec -> всередині опису з'єднання) аналогічні характеристикам підмодулів.

Наступні приклади з'єднань використовують два типу каналів – Ethernet100 і Backbone. У коді показано синтаксис призначення параметрів cost (вартість) і length (довжина), а також зазначено рядок візуалізації (і інші властивості):

```
a.g++ <--> Ethernet100 <--> b.g++;
a.g++ <--> Backbone {cost=100; length=52km; ber=1e-8;} <--> b.g++;
a.g++ <--> Backbone {@display("ls=green,2");} <--> b.g++;
```

При використанні вбудованих в OMNeT++ типів каналів, ім'я типу може бути опущено; воно буде визначено виходячи з набору параметрів

```
a.g++ <--> {delay=10ms;} <--> b.g++;
a.g++ <--> {delay=10ms; ber=1e-8;} <--> b.g++;
a.g++ <--> {@display("ls=red");} <--> b.g++;
```

Якщо зазначено параметри Datarate або ber, то мається на увазі стандартний тип каналу ned.Dataratechannel. А, якщо ні, то присутність delay або disabled списку параметрів буде вказувати на тип ned.Delaychannel; а якщо ні, то використовується тип ned.Idealchannel. Якщо інші параметри зазначені в з'єднанні без явної вказівки типу каналу, то буде видано помилку.

Параметри з'єднань, аналогічно параметрам підмодулів, можуть бути призначені за допомогою шаблонів ініціалізації, хоча в цьому випадку набагато складніше досягтися збігу шаблону з параметрами, що знижує зручність використання. Канал може бути визначений іменем вихідних воріт у з'єднанні з іменем каналу; ім'я каналу завжди channel. Це проілюстроване на наступному прикладі:

```
module Queueing
{
  parameters:
      source.out.channel.delay = 10ms;
      queue.out.channel.delay = 20ms;
  submodules:
      source: Source;
      queue: Queue;
      sink: Sink;
  connections:
      source.out --> ned.Delaychannel --> queue.in;
      queue.out --> ned.Delaychannel <--> sink.in;
}
```

Якщо використовується двунаправлений зв'язок, то необхідно задавати параметри в обидва напрямки каналу окремо:

```
network Network
{
  parameters:
    hosta.g$o[0].channel.datarate = 100Mbps; // the A -> B connection
    hostb.g$o[0].channel.datarate = 100Mbps; // the B -> A connection
    hosta.g$o[1].channel.datarate = 1Gbps;   // the A -> C connection
    hostc.g$o[0].channel.datarate = 1Gbps;   // the C -> A connection
  submodules:
    hosta: Host;
    hostb: Host;
    hostc: Host;
  connections:
    hosta.g++ <--> ned.Dataratechannel <--> hostb.g++;
    hosta.g++ <--> ned.Dataratechannel <--> hostc.g++;
}
```

Крім того, це не завжди легко з'ясувати, які індекси воріт готови до з'єднань, які ви прагнете настроїти. Якщо об'єкти з'єднання можуть бути дані імена, щоб перевизначити ім'я за замовчуванням "канал", щоб зробити їх легше ідентифікованими. Ця функція планується у майбутніх Omnet ++ релізахОб'єкту каналу за замовчуванням дається ім'я channel. Також можна вказати ім'я каналу в явному виді або перевизначити ім'я за замовчуванням для кожного типу каналу. Метою створення користувацьких імен каналів є спрощення використання їх параметрів з ini-файлів.

Синтаксис опису каналу в з'єднанні аналогічний синтаксису підмодулів: після імені через двокрапку вказується тип каналу (name: type). Тому що й ім'я, і тип не є обов'язковими, то після імені завжди ставиться двокрапка, навіть якщо тип каналу поки не задається.

Наприклад:

```
r1.pppg++ <--> eth1: Ethernetchannel <--> r2.pppg++;
a.out --> foo: {delay=1ms;} --> b.in;
a.out --> bar: --> b.in;
```

Якщо ім'я каналу явно не зазначено, то воно береться з параметра @defaultname опису типу каналу.

```
channel Eth10G extends ned.Dataratechannel like Ieth
{
    @defaultname(eth10G);
}
```

Існує виключення у використанні @defaultname. Якщо тип каналу задається рядком **.channelname.liketype= в ini-файлі, то ім'я каналу, зазначено в @defaultname не може використовуватися як ім'я каналу channelname у цій конфігурації тому, що тип каналу буде відомий тільки в результаті використання цього конфігураційного рядка. Щоб проілюструвати цю проблему, розглянемо тип каналу Eth10G і модуль, що містить наступне з'єднання:

```
r1.pppg++ <--><> like Ieth <--> r2.pppg++;
```

Потім розглянемо наступнийі-Файл:

```
**.eth10G.typename = "Eth10G"    # Won't match! The eth10G name would come from
                                 #  the Eth10G type - catch-22!
**.channel.typename = "Eth10G"  # OK, as lookup assumes the name "channel"
**.eth10G.datarate = 10.01Gbps   # OK, channel already exists with name "eth10G"
```

4.9 Приклади множинних з'єднань

Прості програмні конструкції мови NED (петлі, умовні розгалуження) дозволяють створювати множинні з'єднання.Приклад створення ланцюжка модулів:

```
module Chain
    parameters:
        int count;
    submodules:
        node[count] : Node {
            gates:
                port[2];
        }
    connections allowunconnected:
        for i = 0..count-2 {
            node[i].port[1] <--> node[i+1].port[0];
        }
}
```

Приклад побудови бінарного дерева:

```
simple Binarytreenode {
    gates:
        inout left;
        inout right;
        inout parent;
}

module Binarytree {
    parameters:
        int height;
```

```
    submodules:
        node[2height-1]: Binarytreenode;
    connections allowunconnected:
        for i=0..2(height-1)-2 {
            node[i].left <--> node[2*i+1].parent;
            node[i].right <--> node[2*i+2].parent;
     }
}
```

Тут не всі ворота модулів будуть підключені. За замовчуванням, не підключені ворота спровокують повідомлення про помилку під час виконання моделювання, але за рахунок модифікатора allowunconnected у прикладі дана помилка відключена.

Отже, відповідальність за те, щоб не посилати пакети на непідключені ворота покладає на прості модулі.Умовні з'єднання можуть бути використані для генерації випадкових топологій, наприклад код, що випливає, генерує випадковий підграф повного графа:

```
module Randomgraph {
        parameters:
        int count;
        double connectedness; // 0.0<x<1.0
      submodules:
        node[count]: Node {
          gates:
              in[count];
              out[count];
          }
       connections allowunconnected:
          for i=0..count-1, for j=0..count-1 {
              node[i].out[j] --> node[j].in[i]
              if i!=j && uniform(0,1)<connectedness;
          }
}
```

Є кілька підходів для створення складних топологий, що мають регулярну структуру, зміст яких полягає у використанні шаблонів.

Даний шаблон ухвалює підмножину з'єднань повного графа. Умова використовується для одержання необхідних взаємозв'язків з повного графа:

```
for i=0..N-1, for j=0..N-1 {
    node[i].out[...] --> node[j].in[...] if condition(i,j);
}
```

Складений модуль Randomgraph (представлений раніше) являє приклад цього шаблону, однак шаблон може згенерувати будь-який граф, у якому можна сформулювати відповідну умову condition(i,j). Наприклад, при створенні деревоподібної структури, умова буде повертати, чи є вузол *j* дочірнім вузлом *i*-му або навпаки.

Хоча цей шаблон носить досить загальний характер, його використання може бути утруднено, якщо число вузлів N є високим, і граф має мало з'єднань. Наступні два шаблони не мають цього недоліку.

Наступний шаблон перебирає всі вузли й створює необхідні з'єднання для кожного з них. Це записується таким чином:

```
fori=0..Nnodes, forj=0..Nconns(i)-1 {
    node[i].out[j] --> node[rightnodeindex(i,j)].in[j];
}
```

Використання цього шаблону залежить від того, як легко реалізується функція rightnodeindex(i,j).

Наступний шаблон перераховує всі з'єднання усередині циклу:

```
for i=0..Nconnections-1 {
   node[leftnodeindex(i)].out[...] --> node[rightnodeindex(i)].in[...];
}
```

Ця модель може бути використана, якщо топографічні (mapping) функції leftnodeindex(i) і rightnodeindex(i) у достатньому ступені сформульовані.

Модуль Chain являє приклад такого підходу, при якому функції перетворення реалізовані надзвичайно просто: leftnodeindex(i) = i і rightnodeindex(i) = i+1. Шаблон також може бути використаний для створення випадкової підмножини повного графа з фіксованим числом з'єднань.

У випадку нерегулярних структур, де з перерахованих вище шаблонів жоден не може бути використаний, можна вдатися до перерахування всіх можливих з'єднань.

4.10 Параметричні підмодулі й типи з'єднань

Тип підмодуля може бути зазначений у строковому (String) параметрі складеного модуля, або будь-яким строковим вираженням. Синтаксис використовує ключове слово like. Наприклад:

```
network Net6
{
    parameters:
        string nodetype;
    submodules:
        node[6]: <nodetype> like Inode {
            address = index;
        }
    connections:
        ...
}
```

У прикладі створюється вектор підмодулів, тип його елементів буде залежити від параметра nodetype. Наприклад, якщо параметру nodetype буде привласнено значення Sensornode, те вектор модулів node буде складатися з вузлів датчиків Sensornode, за умови, що такий тип модуля існує й може бути підставлений. Це значить, що повинен існувати інтерфейсний модуль Inode, який реалізується модулем Sensornode.

Вираз (дженерик) записується між кутовими дужками, він може використовувати параметри батьківського модуля й раніше певних підмодулів і повинен нести строкове значення. Розглянемо на прикладі наступного коду:

```
network Net6{
    parameters:
        string nodetypeprefix;
        int variant;
    submodules:
        node[6]: <nodetypeprefix + "Node" + string(variant)> like Inode {
            ...
        }
}
```

Пов'язаний з описаною мережею код мовою NED:

```
module interface Inode{
    parameters:
        int address;
    gates:
        inout port[];
}
module Sensornode like Inode{
    parameters:
      int address;
     ...
    gates:
      inout port[ ];
      ...
}
```

Синтаксис <nodetype> like Inode не дозволяє задавати різні типи для різних індексів вектора. Наступний синтаксис краще підходить для векторів підмодулів. Вираз між кутовими дужками може бути опущений <>:

```
module Node{
  submodules:
    nic: <> like Inic;  // type name expression left unspecified
...
}
```

У цьому випадку очікується, що ім'я типу підмодуля буде визначено за допомогою шаблону Typename. Шаблон Typename виглядає як шаблон оголошення параметрів підмодулів, а замість імені параметра задається ключове слово Typename. Також шаблон оголошення Typename можна задати в конфігураційному файлі. У мережі, що моделюється, яка використовує вузли типу Node, шаблон оголошення Typename буде виглядати таким чином:

```
network Network{
  parameters:
      node[*].nic.typename = "Ieee80211g";
  submodules:
       node: Node[100];
}
```

Значення за замовчуванням для типу можна задавати між кутовими дужками; воно буде використано, якщо не був заданий тип:

```
module Node{
   submodules:
       nic: <default("Ieee80211b")> like Inic;
...
}
```

Тут повинен бути рівно один модуль, тип якого заданий під іменем Ieee80211b, і який реалізує інтерфейс Inic, а якщо ні, то буде видано повідомлення про помилку. Якщо є два або більш однакових типів, то необхідно позбутися неоднозначності, указавши повне ім'я типу модуля, яке містить у собі ім'я пакета:

```
module Node{
   submodules:
      nic: <default("acme.wireless.Ieee80211b")> like Inic; // made-up name
...
}
```

Параметричні типи з'єднань працюють аналогічно параметричним типам підмодулів і мають аналогічний синтаксис. У прикладі використовується параметр батьківського модуля:

```
a.g++ <--> <channeltype> like Imychannel <--> b.g++;
a.g++ <--> <channeltype> like Imychannel {@display("ls=red");} <--> b.g++;
```

У виразі можуть використовуватися змінні циклу, параметри батьківського модуля, а також параметри підмодулів (наприклад, host[2].channeltype).

Вираз, що задає тип, також може бути відсутнім, тоді тип, повинен бути зазначений за допомогою шаблону оголошення Typename:

```
a.g++ <--><>likeimychannel<-->b.g++;
a.g++ <--><>likeimychannel {@display("ls=red");} <-->b.g++;
```

Значення за замовчуванням задаються аналогічно типам модулів:

```
a.g++ <--><default("Ethernet100")> like Imychannel <--> b.g++;
a.g++ <--><default(channeltype)> like Imychannel <--> b.g++;
a.g$o[0].channel.typename = "Ethernet1000";  // A -> B channel
b.g$o[0].channel.typename = "Ethernet1000";  // B -> A channel
```

4.11 Анотація метаданих (Властивості)

Використання властивостей (Properties) дозволяє додавати додаткову інформацію в NED елементи. Деякі властивості інтерпретуються ядром моделювання мови NED, інші властивості читаються й використовуються імітаційною моделлю або виконують функції підказок для інструментів редагування мови NED.

Властивості закріплені за типами, це значить, що не можна використовувати невідомі властивості в екземплярах типів. Усі екземпляри модулів, з'єднань, параметрів одного класу мають ідентичні властивості. У прикладі показаний синтаксис оголошення декількох елементів мови NED:

```
@namespace(foo);  // file property
module Example{
    parameters:
        @node;   // module property
        @display("i=device/pc");   // module property
        int a @unit(s) = default(1); // parameter property
    gates:
        output out @loose @labels(pk);  // gate properties
    submodules:
        src: Source {
            parameters:
                @display("p=150,100");  // submodule property
                 count @prompt("Enter count:"); // adding a property to a parameter
            gates:
                 out[] @loose;  // adding a property to a gate
           }
      ...
    connections:
        src.out++ --> { @display("ls=green,2"); } --> sink1.in; // connection prop.
        src.out++ --> Channel { @display("ls=green,2"); } --> sink2.in;
```

```
}
```

Іноді корисно мати кілька властивостей з однаковими іменами, наприклад, для оголошення множинних статистичних характеристик, одержуваних за допомогою простого модуля. Це можливість реалізується шляхом додавання до імені властивості іменованого індексу у квадратних дужках, як eed або jitter у прикладі.

```
simple App {
   @statistic[eed](title="end-to-end delay of received packets";unit=s);
   @statistic[jitter](title="jitter of received packets");
}
```

У прикладі оголошено дві статистичні характеристики під одним іменем властивості @statistic: @statistic[eed] і @statistic[jitter]. Властивості і їх значення в круглих дужках використовуються для введення додаткової інформації, як, наприклад, була зазначена повна назва статистики (title="...") або одиниця виміру (unit=s). Індекси властивостей доступні з інтерфейсу розробки C++, можна запросити список використованих індексів для властивості @statistic, і будуть видані значення eed і jitter.

```
simple Dummy {
   @foo[1](what="apples";amount=2);
   @foo[2](what="oranges";amount=5);
}
```

У прикладі було показано використання числових індексів, значення яких можна перевизначати через спадкування:

```
simple Dummyext extends Dummy {
    @foo[2](what="grapefruits"); // 5 grapefruits instead of 5 oranges
}
```

Властивості можуть містити дані, які задаються в круглих дужках; модель даних є досить гнучкою. Властивість може зовсім не містити значень або мати одне значення:

```
@node;
@node(); // same as @node
@class(Ftpapp2);
```

Властивості можуть містити списки:

```
@foo(Sneezy,Sleepy,Dopey,Doc,Happy,Bashful,Grumpy);
```

Вони можуть містити пари ключ-значення, розділені між собою крапкою з комою:

```
@foo(x=10.31; y=30.2; unit=km);
```

У парах ключ-значення, кожне значення може являти собою список (елементи розділяються комами):

```
@foo(coords=47.549,19.034;labels=vehicle,router,critical);
```

Наведені вище приклади є окремими випадками загальної моделі даних. Відповідно до моделі даних, властивості містять список пар ключ-значення, розділені крапкою з комою. Елементи списку значень, розділяються комами.

Значеннями можуть бути слова, числа, строкові константи й деякі інші символи, але не довільні рядки. Щораз, коли синтаксисом не допускається деяке значення, воно повинно бути укладене в подвійні лапки. Їхнє використання не впливає на саме значення, оскільки інтерпретатор автоматично забирає один рівень лапок; Таким чином, записи `@class(TCP)` і `@class("TCP")` аналогічні. Якщо логікою програми потрібно ввести в рядку лапки, то їх треба екранувати спеціальним символом: `@foo ( "\" деякий рядок \ "")`.

Існує домовленість, що можна використовувати властивості для установки тегів на NED елементи; наприклад, властивість `@host` може бути використано для маркірування всіх типів модулів, які представляють різні хости. Ця властивість може бути розпізнана, наприклад, інструментами редагування, або кодом дослідження топології в імітаційній моделі. Домовленість для такої властивості-маркера полягає в тому, що які-небудь додаткові дані в ньому (тобто в межах дужок) ігнорується, за винятком одного слова `false`, яке виключає ця властивість. Таким чином, будь-яка імітаційна модель або інструмент, який інтерпретує властивості, повинні обробляти всі наступні форми як еквівалентні `@host`: `@host()`, `@host(true)`, `@host(anything-but-false)`, `@host(a=1;b=2)`; а `@host(false)` слід інтерпретувати як відсутність тегу `@host`.

Коли успадковується NED тип, використовується тип модуля в якості підмодуля або використовується тип каналу для з'єднання, можна додавати нові

властивості в модуль або канал, або їх параметрам і воротам, і можна змінювати існуючі властивості.

При зміні властивості, нова властивість поєднується зі старою відповідно до декількох простих правил. Нові ключі просто додаються. Якщо такий ключ вже існує в старій властивості, елементи його списку значень перезаписують елементи у відповідній позиції старої властивості. Одиночний дефіс як елемент списку значень видаляє елемент у відповідній позиції. Приклад порядку завдання властивостей:

Було	@prop
Задали	@prop(a)
Результат	@prop(a)

Було	@prop(a,b,c)
Задали	@prop(,-)
Результат	@prop(a,,c)

Було	@prop(foo=a,b)
Задали	@prop(foo=A,,c;bar=1,2)
Результат	@prop(foo=A,b,c;bar=1,2)

Наведені вище правила злиття не виконуються для певних стандартних властивостей. Наприклад, оголошена властивість @unit не може бути перевизначене в майбутньому, а @display поєднується за особливим правилом.

4.12 Пакети

Зберігати всі NED файли в одному каталозі підходить тільки для невеликих проектів моделей. Коли проект ускладнюється, рано чи пізно стає необхідним ввести структуру каталогів, і сортувати в них NED файли. Мова NED підтримує дерева каталогів з NED файлами, які називаються пакетами (Package). Пакети вирішують проблему конфлікту імен, перед іменем типу ставиться ім'я пакета, як у мові Java.

Перед початком симуляції необхідно вказати ядру моделі кореневу папку пакета. Мова NED обійде все дерево підкаталогів і завантажить NED файли із усіх вкладених папок. Можна використовувати кілька дерев NED каталогів, тоді кореневу папку кожного необхідно вказати в змінній NEDPATH. Її можна задати такими способами: у якості змінної оточення NEDPATH, як опцію конфігурації ned-path або як параметр командного рядка із ключем (-n).

Папки у вихідному дереві NED каталогу відповідають пакетам. Якщо є NED файли в папці <root>/a/b/c (де <root> зазначений у змінній NEDPATH), то відповідне йому ім'я пакета буде a.b.c. Ім'я використовуваного пакета повинне бути явно оголошено у верхній частині NED файлу таким чином:

```
package a.b.c;
```

Ім'я пакета, що зберігається в деякій папці, і оголошеного пакета в програмі повинні збігатися. За згодою імена пакетів пишуться в нижньому регістрі й починаються з назви проекту (myproject) або являють собою записане у зворотному порядку доменне ім'я перед назвою (org.example.myproject). В останньому випадку буде пройдено дерево каталогів, щоб знайти потрібні файли. У самому верхньому каталозі розміщають файл package.ned, який представляє весь пакет. Наприклад, коментарі в package.ned розглядаються в якості документації пакета. Крім того, властивість @namespace (простір імен) у файлі package.ned впливає на всі NED файли в цьому каталозі й усі каталоги глибше.

Файл package.ned верхнього рівня може бути використаний для позначення кореневого пакета, який корисний для усунення декількох рівнів порожніх каталогів, які створюються через правило іменування пакетів. Наприклад, якщо в проекті необхідно мати всі типи NED файлів у рамках пакета org.example.myproject, але створювати дерево вкладених каталогів з іменами org, example і myproject з деяких причин не потрібно, те можна покласти файл package.ned у вихідному кореневому каталозі пакета й вставити оголошення пакета org.example.myproject. Це приведе до того, що деякий каталог foo, який перебуває в корені, повинен інтерпретуватися як пакет org.example.myproject.foo і всі NED файли, що містяться в ньому, повинні вказувати дане ім'я пакета. Тільки кореневий файл package.ned може визначати пакет, файли package.ned у підкаталогах повинні дотримуватися його політики.

Як приклад розглянемо пакет INET Framework, який містить сотні NED файлів і кілька десятків пакетів. Структура каталогів виглядає таким чином:

```
INET/
    src/
        base/
        transport/
            tcp/
            udp/
            ...
        networklayer/
        linklayer/
        ...
    examples/
        adhoc/
        ethernet/
        ...
```

Підкаталоги src і examples позначаються як папки вихідних кодів NED, тому NEDPATH буде наступним, за умови, що INET розпакований у папку /home/user:

/home/user/INET/src; /home/user/INET/examples

Обидва підкаталоги src і examples містять package.ned файли для визначення кореневого пакета:

```
// INET/src/package.ned:
package inet;
// INET/examples/package.ned:
package inet.examples;
```

У свою чергу, інші NED файли додержуються визначення пакета, заданому в package.ned:

```
// INET/src/transport/tcp/TCP.ned:
package inet.transport.tcp;
```

Було сказано, що пакети можуть використовуватися для відмінності однаково названих NED типів. Ім'я типу, що включає в запис ім'я пакета (a.b.c.Queue для модуля черги Queue у пакеті a.b.c) називається повним іменем; якщо ім'я пакета опущено (Queue), то такий запис називається простим іменуванням.

Вказівки простого імені не достатньо, щоб однозначно ідентифікувати тип. Звернутися до існуючого типу можна такими способами:

1. Задавши повне ім'я типу. Такий спосіб часто буває громіздким.
2. Імпортувавши тип. У цьому випадку простого імені буде досить.
3. Якщо тип перебуває в тому ж пакеті, то його нетреба імпортувати. На нього можна посилатися по простому імені.

Щоб імпортувати тип, спочатку вказують ключове слово import, потім указують повне ім'я типу або задають шаблон підстановки. У шаблонах підстановки одній зірочці "*" відповідає будь-яка послідовність символів, що не містить крапку, а двом зірочкам "**" відповідає будь-яка послідовність символів, яка може містити крапку.

У прикладі, кожна з наступних рядків імпортує тип calledinet.protocols.networklayer.ip.Routingtable:

```
import inet.protocols.networklayer.ip.Routingtable;
```

```
import inet.protocols.networklayer.ip.*;
import inet.protocols.networklayer.ip.Ro*Ta*;
import inet.protocols.*.ip.*;
import inet.**.Routingtable;
```

Якщо директива import явно вказує тип по точному повному імені, то такий тип повинен існувати, а якщо ні, то буде видана помилку. Директива import, яка містить шаблони підстановки допускає невідповідність зазначеного імені ніякому існуючому NED типу, але тоді буде сгенерировано попередження.

До внутрішніх типів не можна звертатися до типу, що ззовні обрамляє.

Використання ключового слово like для з'єднань відрізняється від його використання з підмодулями, коли ім'я типу береться зі строкового виразу. Імпортування через ключове слово import не корисно тут. Під час написання NED файлу ще невідомо, які NED типи будуть у ньому підходящими для підключення, тому їх заздалегідь підключити не можна.

З повними іменами проблем ніяких ні, але із простими іменами проблема вирішується по-різному. У коді NED визначається, який інтерфейс реалізовують типи модулів або каналів (про прикладу Inode), а потім збираються типи, які мають зазначене просте ім'я й реалізовують даний інтерфейс. У групі повинен бути рівно один такий тип, який буде використовуватися в наслідок. Якщо такого типу немає в групі або існує більш ніж один тип, то буде видано повідомлення про помилку.

Розглянемо наступний приклад:

```
module Mobilehost{
   parameters:
      string mobilitytype;
   submodules:
      mobility: <mobilitytype> like Imobility;
      ...}
```

Припустимо, що наступні модулі реалізують інтерфейс модуля Imobility: inet.mobility.Randomwalk, inet.adhoc.Randomwalk, inet.mobility.Massmobility. Також припустимо, що є такий тип inet.examples.adhoc.Massmobility, який не реалізує даний інтерфейс.

Таким чином, якщо параметр mobilitytype = "Massmobility", то буде обрано модуль inet.mobility.Massmobility, а інший тип Massmobility не заважає вибору. Однак, якщо змінна mobilitytype = "Randomwalk", то буде видано помилку, тому що буде два типи, що відповідають Randomwalk. Кожні з типів

Randomwalk як і раніше можуть бути використані, але тоді потрібно явно вибрати один з них, задаючи повне ім'я, що включає ім'я пакета: mobilitytype = "inet.adhoc.Randomwalk"

Якщо всі NED файли перебувають в одному каталозі, записаному в NEDPATH, то декларацію пакета (import) можна не писати. Уважається, що ці файли перебувають у пакеті за замовчуванням (default package).

4.13 Висновки

У розділі розглянуто мову NED, яка дозволяє описувати структуру та топологію мереж та підмереж. Показана структура та загальні розділи мови NED. Розглянуто прості модулі, які являють собою основу роботи імітаційної моделі разом із календарем подій. У свою чергу, складні модулі виконують функцію опису топології мережі. Наведено декілька стандартних типів з'єднань модулів компонентами-каналами. Показано яким чином модулі набувають множинні з'єднання.

У кінці глави розглядаються питання анотації метаданих та групування розроблений модулів у пакети.

Контрольні питання.

1. Базові позначення мови програмування NED.
2. Навести приклад мережі, топологію якої необхідно реалізувати мовою NED.
3. Призначення розділу декларації connections.
3. Прості модулі (Simple modules).
4. Складені модулі. Оголошення складеного модуля.
5. Призначення розділів parameters і gates.
6. Канали зв'язку модулів. Типи каналів та параметри.
7. Стандартний тип Delaychannel.
8 Стандартний тип Dataratechannel.
9. Надання параметрів (Parameters) модулю. Параметр packetlength.
10 Загальні властивості модифікатору volatile.
12 Використання воріт (Gates).
13 Підмодулі. Масиви підмодулів. Ініціалізація підмодулів.
14 З'єднання (Connections) модулів та підмодулів. Множинні з'єднання.
15 Параметричні підмодулі й типи з'єднань.
16 Анотація метаданих (Властивості).
17 Групування модулів у пакети.

5. ПРОСТІ МОДУЛІ

Прості модулі є активними компонентами моделей. Прості модулі розробляються на C++ з використанням бібліотеки класів OMNeT++. У цій главі міститься коротке введення до дискретного моделювання подій у цілому, пояснюється, яким чином його концепції реалізовані в OMNeT++ і дається короткий огляд і практичні поради про те, як проектувати код простих модулів.

5.1 Концепції моделювання

Дискретна система подій, discrete event simulation (DES), являє собою систему, де зміни станів (подій) відбуваються в дискретні моменти часу, а події мають нульову тривалість. Передбачається, що нічого (тобто нічого цікавого) не відбувається між двома послідовними подіями, тобто, ніяких змін стану не відбувається в системі між подіями, на відміну від безперервних систем, де зміни станів безперервні. Системи, які розглядають як дискретні системи подій, можна моделювати за допомогою дискретного моделювання подій DES.

Комп'ютерні мережі, як правило, розглядаються як дискретні системи подій, де є, наприклад події, що випливають на:

- початок пакетної передачі;
- кінець пакетної передачі;
- витікання терміну дії тайм-ауту повторної передачі.

Це значить, що між двома такими подіями, як початок передачі пакета (start of a packet transmission) і кінець передачі пакета (end of a packet transmission), нічого цікавого не відбувається. Тобто пакет залишається в стані передачі. Визначення "цікавих" подій і станів завжди залежить від намірів моделювання. Якби було завдання передавати окремі біти, то в сукупність подій моделі були б додані наступні події: початок передачі бітів (*start of bit transmission*) і кінець передачі бітів (*end of bit transmission*).

Час, коли події відбуваються, часто називають часовою міткою (*event timestamp*) події. OMNeT++ використовує термін час прибуття (*arrival time*), тому що в бібліотеці класів слово «*timestamp*» зарезервовано для встановленого користувачем атрибута в класі подій. Час у рамках моделі часто називають віртуальним часом моделювання (*simulation time*), який не збігається з реальним часом тривалості виконання програмиДискретне моделювання подій містить безліч майбутніх подій у структурі даних FES (Future Set Event) або

FEL (Future Event List). Такого роду моделі звичайно працюють у відповідності з наступним псевдокодом:

```
initialize – this includes building the model and inserting initial events to FES
while (FES not empty and simulation not yet complete)
{
    retrieve first event from FES
    t:= timestamp of this event
    process event
    (processing may insert new events in FES or delete existing ones)
}
finish simulation (write statistical results, etc.)
```

Етап ініціалізації звичайно створює структури даних, які представляють імітаційну модель, викликає певний користувачем деякий код ініціалізації й вставляє початкові події в FES для початку моделювання. Стратегії ініціалізації можуть значно відрізнятися в різних моделей.

Наступний цикл використовує події з FES і обробляє їх. Події обробляються в строгому порядку згідно зі значеннями часових міток для підтримки причинно-наслідкового зв'язку, тобто, щоб гарантувати, що поточна подія не може впливати на більш ранні події.

Обробка події містить у собі виклик, що поставляється користувачем коду. Наприклад, у моделі комп'ютерної мережі обробка події закінчення тайм-ауту (timeout expired) може містити в собі повторне відправлення копії мережного пакета, відновлення лічильника повторів, планування ще одного тайм-ауту, і так далі. Користувацький код може також видаляти події з FES, наприклад, при скасуванні тайм-ауту.

Моделювання зупиняється, коли більше не залишилося подій (це рідко буває на практиці), або коли моделі немає необхідності виконуватися далі, тому що час моделювання або час центрального процесора досяглися заданої межі, або тому, що статистика досяглася бажаної точності. Тоді перед виходом із програми користувач, як правило, прагне записати статистику у вихідні файли.

OMNeT++ використовує повідомлення для вистави події. Для всіх практичних цілей існує клас Cevent, який успадковує cmessage і є внутрішнім стосовно ядра моделювання.

Повідомлення представлені екземплярами класу cmessage і його підкласами. Повідомлення передаються від одного модуля іншому. Це значить, що місцю, у якому буде відбуватися подія, буде відповідати модуль призначення даного повідомлення (message's destination module), а значенню

часу моделі, коли відбудеться подія, – час приходу повідомлення (arrival time). Такі події, як "Час очікування минув", реалізуються за допомогою модуля відправлення повідомлення самому собі.

Події витягуються з FES у порядку значень часу прибуття, щоб підтримувати причинно-наслідковий зв'язок. Для вибору повідомлення застосовуються наступні правила:

1. Повідомлення з більш раннім часом прибуття виконується в першу чергу.

2. Якщо час прибуття дорівнюють один у одного, повідомлення з більш високим пріоритетом планування (scheduling priority), тобто з меншим числовим значенням, виконується в першу чергу.

3. Якщо пріоритети однакові, то відправлене раніше повідомлення виконується в першу чергу.

Пріоритетом планування є привласнене користувачем ціле значення відповідному до атрибута повідомлення.

Поточний час моделювання може бути отриманий за допомогою функції simtime().

Час моделювання в OMNeT++ представлений C++ типом simtime_t. Клас Simtime зберігає час моделювання у вигляді 64-бітного цілого числа, використовуючи десяткове уявлення з фіксованою комою. Точність управляється змінною глобальної конфігурації scale exponent, таким чином, екземпляри Simtime мають однаковий масштаб. Показник ступеня може бути обраний від -18 до 0 (секунди).

Немає неявного перетворення з Simtime в double в основному, тому що це буде суперечити перевантаженню арифметичних операцій класу Simtime. Для такого перетворення використовується метод dbl() із класу Simtime або макрос SIMTIME_DBL(). Для того, щоб зменшити потребу у використанні dbl(), деякі функції й методи мають перезавантаженні варіанти для безпосереднього приймання Simtime, наприклад fabs(), fmod(), div(), exponential(), normal().

Інші корисні методи Simtime: Str() - повертає значення у вигляді рядка; parse() - перетворить рядок в Simtime; raw() - повертає базове значення в int64; getscaleexp() - повертає глобальний показник масштабу; iszero() – перевіряє час моделювання на рівність 0; getmaxtime() - повертає максимальний час моделювання, представлений поточним масштабом ступеню. Нульовий і максимальний час моделювання доступний за допомогою макросів SIMTIME_ZERO і SIMTIME_MAX.

```
// 340 microseconds in the future, truncated to millisecond boundary
simtime_t timeout = (simtime() + Simtime(340, SIMTIME_US)).trunc(SIMTIME_MS);
```

Реалізація FES є вирішальним чинником при виконанні дискретного моделювання подій. В OMNeT++ дискретна система подій FES реалізована у вигляді бінарної купи (binary heap), найбільше широко використовуваної структури даних для подібних цілей. FES реалізована в класі cmessageheap.

5.2 Компоненти, прості модулі, канали

У OMNeT++ імітаційні моделі складаються з модулів і з'єднань. Модулі можуть бути простими або складовими. Прості модулі є активними компонентами в моделі, і їх поведінка задається користувачем у якості коду C++. З'єднання асоціюються з об'єктами каналів (Channel). Канальні об'єкти інкапсулюють поведінку каналів: моделювання поширення сигналу, часу передачі, появи помилок і т.п. Канали також можуть бути запрограмовані користувачем на C ++.

Модулі й канали представлені класами cmodule і cchannel відповідно, які походять від класу ccomponent.

Користувач визначає прості типи модулів за допомогою створення підкласів csimplemodule. Складені модулі інкапсулюються у cmodule, хоча користувач може змінити ім'я предка за допомогою @CLASS в NED файлі. Також можна використовувати C++ клас простого модуля (тобто похідний від csimplemodule) для складеного модуля.

Клас cchannel успадковується трьома вбудованими типами каналів: cidealchannel, cdelaychannel і cdataratechannel. Користувачем можуть бути створені нові типи каналів шляхом створення підкласів від cchannel або будь-якого іншого класу каналу. Взаємозв'язок цих класів ілюструється на рис.60.

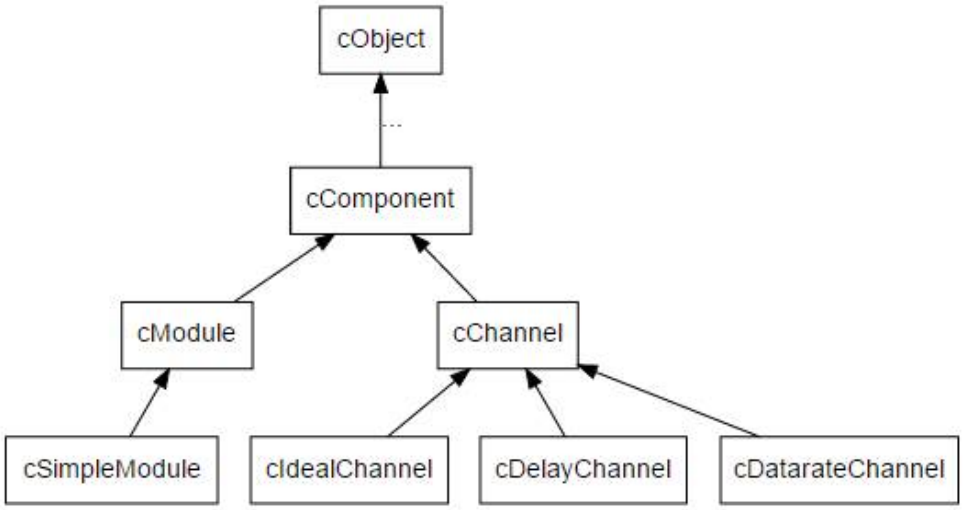

Рис.60 - Спадкування компонента, модуля й каналів класів

Прості модулі й канали можуть бути перепрограмовані шляхом перевизначення деяких функцій, які, у свою чергу могли бути оголошені в ccomponent, загальному базовому класі каналів і модулів.

Клас ccomponent має наступні функції, призначені для перевизначення в підкласах:

– Initialize() – надає місце для коду ініціалізації й викликається після того, як OMNeT++ створить мережу, усі модулі і їх з'єднання відповідно до визначень;

– finish() – викликається, коли моделювання завершується успішно. Його рекомендується використовувати для запису зведеної статистики.

В OMNeT++ події відбуваються усередині простих модулів. Прості модулі інкапсулюють C ++ код, який генерує події й реагує на них, іншими словами, реалізує поведінку моделі.

Для того, щоб визначити динамічну поведінку простого модуля, необхідно перевизначити одну з наступних функцій:

– handlemessage (cmessage *msg). Викликається з повідомленнями у якості параметра щораз, коли модуль одержує повідомлення. Функція призначена для обробки отриманого повідомлення, після чого вона повертає керування. Час моделювання ніколи не змінюється усередині виклику handlemessage (), а тільки між ними.

– activity() запускається як сопрограма на початку моделювання й працює до кінця імітації. Повідомлення надходять після виклику функції receive(). Час моделювання змінюється усередині виклику receive().

Модулі, написані з activity() і handlemessage(), можуть вільно поєднуватись в імітаційній моделі. Як правило, віддається перевага handlemessage(), а не activity(), через масштабованість і з деяких інших практичних причин.

Поведінка каналів також може бути змінена шляхом перевизначення функцій.

5.3 Визначення типів простих модулів

Простий модуль являє собою клас С ++, який повинен бути успадкований від csimplemodule, причому в ньому повинні бути перевизначени одна або кілька віртуальних функцій для завдання його поведінки.

Клас повинен бути зареєстрований в OMNeT++ через макрос Define_Module(). Рядок з Define_Module() завжди повинен бути введений в.cc або .cpp файлі, але не в заголовному файлі .h.

Далі представлений Hellomodule, практично найпростіший модуль, який можна було б написати. Позначимо клас csimplemodule як базовий клас, і введемо рядок Define_Module().

```
// file: Hellomodule.cc
#include <omnetpp.h>
using namespace omnetpp;

class Hellomodule : public csimplemodule
{
 protected:
   virtual void initialize();
   virtual void handlemessage(cMessage*msg);
};

//registermodulecmessageth OMNeT++
Define_Module(Hellomodule);

void Hellomodule::initialize()
{
   EV << "Hello World!\n";
}

void Hellomodule::handlemessage(cMessage*msg)
```

```
{
deletemsg;//justdiscmessagerything we receive
}
```

Для того, щоб була можливість посилатися на цей тип простого модуля в NED файлах, необхідна асоційована NED декларація, яка виглядає таким чином:

```
// file: Hellomodule.ned
simplehellomodule
{
   gates:
      input in;
}
```

Прості модулі інстанцуються не прямо користувачем, а ядром моделювання. Це значить, що не можна створювати довільні конструктори: оголошення повинно бути таким, яке очікує ядро моделювання. На щастя, ця угода є дуже простою: конструктор повинен бути загальнодоступним (public), і не повинен приймати ніяких аргументів:

```
public:
   Hellomodule();  // constructor takes no arguments
```

Клас csimplemodule, у той же час, має два конструктори: csimplemodule() без аргументу й csimplemodule(size_t stacksize), який приймає розмір сопрограмного стека. Перший варіант використовується з функцією handlemessage() простих модулів, а другий – з функцією activity(). Передача нульового розміру стека в останній варіант конструктора викличе handlemessage().

Таким чином визначення конструкторів, що випливають, вірні й вибирають функцію handlemessage(), яка буде використовуватися модулем:

```
Hellomodule::Hellomodule() {...}
Hellomodule::Hellomodule() : csimplemodule() {...}
```

Не буде помилкою вилучити опис конструктора, тому що підходить конструктор, сгенерований компілятором.

У наступному визначенні конструктор вибирає activity() для використання з модулем, задаючи 16384 до сопрограмного стека:

```
Hellomodule::Hellomodule() : csimplemodule(16384) {...}
```

Методи initialize() і finish() оголошені частиною класу ccomponent і надають користувачеві можливість виконання коду на початку моделювання й при успішному його закінченні.

Функція initialize() використовується, як правило, коли не можна поставити код, пов'язаний з моделюванням, у конструктор модуля, тому що імітаційна модель ще налаштовується при запуску конструктора, і багато необхідних об'єктів ще недоступні. Метод initialize() викликається безпосередньо перед симуляцією й починає виконання, коли всі інші об'єкти вже були створені.

Функція finish() використовується для запису статистики й викликається тільки, коли моделювання завершилося нормально. Вона не буде викликана, коли моделювання зупиняється з повідомленням про помилку. Деструктор завжди викликається в самому кінці, незалежно від того, яким чином моделювання було зупинено.

Виходячи з вищевказаних міркувань, угоди про використання цих чотирьох методів наступні:

– Конструктор: Встановлює покажчик членів модуля класу на nullptr. Покладає всі інші завдання ініціалізації на initialize().

– initialize(): Виконує всі завдання ініціалізації: зчитування параметрів модуля, ініціалізація змінних класу, виділення динамічних структур даних операцією new. Також виділяє пам'ять і ініциалізує таймери у вигляді повідомлень самому собі (self-messages), якщо необхідно.

– finish(): Записує статистику. Не слід тут нічого видаляти або скасовувати таймери - усі очищення повинні виконуватися в деструкторі.

– деструктор: Тут треба видаляти все, що було виділено оператором new і усе ще утримується в пам'яті класом модуля. При відісланому повідомленні самому собі (таймер) треба використовувати функцію cancelanddelete(msg). Неправильно просто вилучити повідомлення-таймер у деструкторі, тому що воно могло б бути в плановому списку подій (scheduled events list). Функція cancelanddelete(msg) спочатку перевіряє список подій і скасовує повідомлення перед видаленням, якщо це необхідно.

Omnet ++ виводить список незвільнених об'єктів наприкінці моделювання. Коли імітаційна модель видає повідомлення "undisposed object ...", це вказує на те, що відповідний модуль деструкторів повинен бути виправлений. У якості

тимчасового заходу, ці повідомлення можна сховати шляхом установки параметра конфігурації print-undisposed=false.

Функції модулів initialize() викликаються до обробки першої події, але після того, як початкові події (стартові повідомлення) були поміщені в FES ядром моделювання.

І прості й складені модулі містять функції initialize(). Функція initialize() складеного модуля запускається перед аналогічними функціями його підмодулів.

Порядок викликів функцій finish() зворотний і відбувається, коли цикл обробки подій минувся, і тільки якщо він завершився нормально. Функція finish() не викликається, якщо моделювання завершується з помилкою під час виконання, вона не є гарним місцем для коду очищення пам'яті, який повинен запускатися щораз, коли модуль віддаляється. Функція finish() використовується для збереження статистики, результатів постобробки й інших операцій, які повинні виконуватися тільки при успішному завершенні. Код очищення повинен перебувати в деструкторі.

В імітаційних моделях, де недостатньо одноступінчастої ініціалізації, виконуваної initialize(), можна використовувати багатоступінчасту ініціалізацію. Модулі мають дві функції, які можуть бути перевизначені користувачем:

```
virtual void initialize(intstage);
virtual int numinitstages() const;
```

На початку моделювання для всіх модулів викликається initialize(0), потім initialize(1), initialize(2), і т.д., таким чином відбувається ініціалізація у декілька хвиль. Для кожного модуля, функція numinitstages() повинна бути перевизначена таким чином, щоб вона повертала необхідну кількість етапів ініціалізації. Наприклад, для двоступінчастої ініціалізації numinitstages() повинна повертати число 2, а initialize(int stage) повинна бути реалізована таким чином, щоб обробляти випадки, коли stage=0 і stage=1

5.4 Додавання функціональності в csimplemodule

У цьому розділі обговорюються раніше згадані функції handlemessage() і activity() – члени класу csimplemodule, які повинні бути перевизначені користувачем.

Ідея полягає в тому, що в кожній події (прибутті повідомлення) викликається реалізована користувачем функція. Ця функція, handlemessage(cmessage *msg) являє собою віртуальну функцію-член класу

csimplemodule, яка нічого не робить за замовчуванням - користувач повинен перевизначити її в підкласах і додати код обробки повідомлень.

Функція handlemessage() буде викликатися для кожного повідомлення, яке надходить у модуль. Функція повинна обробити повідомлення й повернути потік відразу ж після цього. Час моделювання потенційно різниться при кожному виклику. Час моделювання не збільшується протягом виклику handlemessage ().

Цикл подій усередині моделі обробляють і activity() і handlemessage() простих модулів відповідно до наступного псевдокоду:

```
while (FES not empty and simulation not yet complete)
{
   retrieve first event from FES
   t:= timestamp of this event
   m:= module containing this event
   if (m works with handlemessage())
            m->handlemessage( event )
   else // m works with activity()
      transferto( m )
}
```

Модулі, що використовують handlemessage() автоматично не запускаються: ядро моделювання створює стартові повідомлення тільки для модулів з activity(). Це значить, що необхідно запланувати повідомлення самому собі у функції initialize(), якщо передбачається використовувати простий модуль із handlemessage(), і почати його роботу без попереднього одержання повідомлення від інших модулів.

Для того, щоб використовувати механізм handlemessage() із простим модулем, необхідно вказати нульовий розмір стека для модуля. Це вказує OMNeT++, що потрібно використовувати handlemessage() і не використовувати activity().

Функції, пов'язані з повідомленнями або подіями, які використовуються в handlemessage():

– сімейство функцій send() використовуються для відправлення повідомлень в інші модулі;

– scheduleat() - планує подію (модуль посилає повідомлення самому собі);

– cancelevent() - вилучає заплановану подію функцією scheduleat().

Не можна використовувати функції send() і wait() в handlemessage(), тому що вони за замовчуванням є сопрограмними й використовуються з activity().

Потрібно додати елементи даних у клас модуля для кожного обсягу інформації, який необхідно зберегти. Ця інформація не може зберігатися в локальних змінних handlemessage(), тому що вони знищуються, коли функція повертає керування. Крім того, інформація не може бути збережена в статичних змінних функції (або класу), тому що вони розподілені між усіма екземплярами класу.

Фрагменти даних, які додаються до класу модуля, як правило, містять у собі таку інформацію:

– стан (наприклад, холостий хід, зайнятість каналу, обрив з'єднання, з'єднання встановлено);

– інші змінні, які належать стану модуля: кількість повторень, черги пакетів і т.д.;

– значення, які витягуються й обчислюються один раз, а потім зберігаються: значення параметрів модуля, індекси воріт, маршрутна інформація і т.д.;

– покажчики на об'єкти повідомлень, створених колись, а потім повторно використані для таймерів, тайм-аутів і т.д.;

– змінні й об'єкти для збору статистики.

Потрібно ініціалізувати ці змінні у функції initialize(). Конструктор не краще місце для цієї мети, тому що він викликається на етапі настроювання мережі, коли модель ще перебуває в стадії побудови, і більшість інформації, яку будуть використовувати, ще не доступно.

Також в initialize() потрібно запланувати початкові події, які виконують перший виклик handlemessage(). Після першого виклику, handlemessage() повинен подбати, щоб були заплановані подальші події для себе таким чином, щоб ланцюжок подій не був порушений. Планування подій не потрібно, якщо поточний модуль повинен тільки реагувати на повідомлення, що приходять із інших модулів.

Випадки, коли handlemessage() переважніше використовувати, замість activity():

1. Коли передбачається, що модуль буде використаний у великих імітаційних моделях за участю декількох тисяч модулів. У таких випадках стек модулів, необхідний activity(), буде споживати занадто багато пам'яті.

2. Для модулів, які не підтримують практично ніякої інформації про стан, таких як знищення пакетів, handlemessage() більш зручно запрограмувати.

3. Для модулів з більшим простором станів і безлічі довільних можливостей переходів в інші стани. Подібні алгоритми важко програмувати з activity(), і найкраще підходить для цього handlemessage(). Це варіант підходить для більшості комунікаційних протоколів

Моделі стеків протоколів у мережах зв'язку, як правило, мають загальну структуру на високих рівнях, тому що в основному всі вони повинні реагувати на події трьох типів: на повідомлення від протоколів більш високого рівня (або додатків), на повідомлення протоколів нижнього шару (від мережі), а також на різні таймери й час очікування (повідомлення самому собі). Це звичайно приводить до наступної схеми вихідного коду:

```
class FooProtocol : public cSimpleModule{
  protected:
    // state variables
    // ...
  virtual void processMsgFromHigherLayer(cMessage *packet);
  virtual void processMsgFromLowerLayer(FooPacket *packet);
  virtual void processTimer(cMessage *timer);
  virtual void initialize();
  virtual void handleMessage(cMessage *msg);
};
// ...
void FooProtocol::handleMessage(cMessage *msg){
  if (msg->isSelfMessage())
    processTimer(msg);
  else if (msg->arrivedOn("fromNetw"))
    processMsgFromLowerLayer(check_and_cast<FooPacket *>(msg));
  else
    processMsgFromHigherLayer(msg);
}
```

Функції processmsgfromhigherlayer(), processmsgfromlowerlayer() і processtimer() використовуються для обробки різних типів пактів і таймерів.

Код для простих генераторів пакетів і стоків, запрограмованих з handlemessage() представлений наступним кодом:

```
PacketGenerator::handleMessage(msg){
  create and send out a new packet;
  schedule msg again to trigger next call to handleMessage;
}
PacketSink::handleMessage(msg){
  delete msg;
}
```

У класі Packetgenerator необхідно перевизначити функцію initialize(), щоб створити й запланувати першу подію.

Наступний простий модуль генерує пакети з експоненційним розподілом часу між вступами:

```
class Generator : public cSimpleModule{
   public:
      Generator() : cSimpleModule() {}
   protected:
      virtual void initialize();
      virtual void handleMessage(cMessage *msg);
};

Define_Module(Generator);
void Generator::initialize(){
   // schedule first sending
   scheduleAt(simTime(), new cMessage);
}
void Generator::handleMessage(cMessage *msg){
   // generate & send packet
   cMessage *pkt = new cMessage;
   send(pkt, "out");
   // schedule next call
   scheduleAt(simTime()+exponential(1.0), msg);
}
```

Для більшої реалістичності пропонується переписати поточний генератор для створення сплесків (пачковості) пакетів, з параметром burstlength.

Додамо деякі елементи даних у клас:

– burstlength визначає кількість пакетів у пачці;

– burstcounter враховує кількість пакетів, які залишилися для відправлення в поточній пачці.

```
class BurstyGenerator : public cSimpleModule{
   protected:
      int burstLength;
      int burstCounter;
   virtual void initialize();
   virtual void handleMessage(cMessage *msg);
```

```
};
Define_Module(BurstyGenerator);
void BurstyGenerator::initialize(){
   // init parameters and state variables
   burstLength = par("burstLength");
   burstCounter = burstLength;
   // schedule first packet of first burst
   scheduleAt(simTime(), new cMessage);
}
void BurstyGenerator::handleMessage(cMessage *msg){
   // generate & send packet
   cMessage *pkt = new cMessage;
   send(pkt, "out");
   // if this was the last packet of the burst
   if (--burstCounter == 0) {
      // schedule next burst
      burstCounter = burstLength;
      scheduleAt(simTime()+exponential(5.0), msg);
   } else {
      // schedule next sending within burst
      scheduleAt(simTime()+exponential(1.0), msg);
   }
}
```

Плюси використання handlemessage():

– споживає менше пам'яті: немає окремого стека, необхідного для простих модулів;

– швидкість: виклик функції відбувається швидше, чим перемикання між сопрограмами.

Мінуси використання handlemessage():

– локальні змінні для зберігання інформації про стан використовуватися не можуть;

– необхідно перевизначати функцію initialize()

Хоча, як правило, handlemessage() переважніше activity(), з activity(), можна програмувати простий модуль так само, як системний процес або потік операційної системи. Вхідне повідомлення (подія) очікується в будь-якій частині коду, можна припиняти виконання протягом деякого часу (часу моделі). При виході з функції activity(), модуль завершує роботу, хоча імітаційна модель може продовжувати виконання за рахунок інших модулів.

Найбільш важливі функції, які використовуються з activity():

– receive() – для одержання повідомлення (події);

– wait() – для припинення виконання на якийсь час (час моделі);

– сімейство функцій send() - для відправлення повідомлень в інші модулі;

– scheduleat() – для планування подій (модуль посилає повідомлення самому собі);

– cancelevent() – для видалення з розкладу події, заданної функцією scheduleat();

– end() – для закінчення виконання даного модуля (вихід з функції activity())

Функція activity() звичайно містить у своєму тілі нескінченний цикл списку викликів wait() або receive().

Основна незручність функції activity() полягає в тому, що вона не масштабується, оскільки кожному модулю потрібний окремий стек для сопрограми

Існує сценарій, при якому використання activity() зручно: коли процес підтримує багато станів, але переходи між ними досить обмежені. Тобто, з будь-якого стану процес може перейти тільки в одне або два інших станів, наприклад, при програмуванні мережного додатка, у якому використовується одне підключення до мережі. Псевдокод додатка, який спілкується із протоколом транспортного рівня може виглядати таким чином

```
activity(){
   while(true) {
        open connection by sending OPEN command to transport layer
        receive reply from transport layer
        if (open not successful){
            wait(some time)
            continue // loop back to while()
        }
        while (there is more to do){
            send data on network connection
            if (connection broken){
               continue outer loop // loop back to outer while()
            }
            wait(some time)
            receive data on network connection
            if (connection broken){
                 continue outer loop // loop back to outer while()
```

```
            }
            wait(some time)
        }
        close connection by sending CLOSE command to transport layer
        if (close not successful){
            // handle error
        }
        wait(sometime)
    }
}
```

Якщо необхідно обробляти кілька з'єднань одночасно, то можна динамічно їх створювати як екземпляри простого модуля, як показано вище.

Якщо функція activity() не містить wait() і містить тільки один виклик receive() на початку нескінченного циклу, то немає ніякого сенсу у використанні activity(), і код повинен бути перенесений в handlemessage(). Тоді тіло нескінченного циклу потрібно помістити в тіло handlemessage(), змінні стани з тіла activity() стануть членами даних у класі модуля, які треба ініціалізувати в initialize(). Приклад:

```
void Sink::activity(){
   while(true) {
      msg = receive();
      delete msg;
   }
}
```

Краще запрограмувати так:

```
void Sink::handleMessage(cMessage *msg){
   delete msg;
}
```

Функція activity() виконується в стеці сопрограм. Сопрограми схожі на потоки, але плануються не превентивно (кооперативна багатозадачність). З однієї сопрограми можна перемкнутися на іншу за допомогою виклику функції transferto(othercoroutine). Тоді поточна сопрограма припиняє працювати, а othercoroutine буде. Пізніше, коли othercoroutine зробить виклик transferto(firstcoroutine), виконання першої сопрограми буде відновлено із точки

виклику transferto(othercoroutine). Повний стан сопрограми, у тому числі локальні змінні, зберігається, у той час як потік виконання перебуває в інших сопрограмах. Це означає, що кожна сопрограма повинна мати свій власний стек процесу, а transferto () викликає перехід від одного стека процесу на іншій.

Сопрограми перебувають у самому центрі OMNeT++, розроблювачеві не завжди потрібно викликати transferto() або інші функції із сопрограмної бібліотеки, і непотрібне опікуватися про реалізацію сопрограмної бібліотеки. Однак важливо розуміти, яким образом цикл подій знаходить у дискретних моделях події, які працюють із сопрограмами. При використанні сопрограм, цикл обробки подій виглядає таким чином:

```
while (FES not empty and simulation not yet complete){
   retrieve first event from FES
   t:= timestamp of this event
   transferto(module containing the event)
}
```

Тобто, коли модуль одержує подію, ядро моделювання передає керування сопрограмі модуля. Коли модуль завершує обробку події, він передає керування назад у ядро моделювання за допомогою виклику функції transferto(main). Прості модулі, що використовують activity(), завантажені на згадку стартовими подіями, вставленими в FES ядром моделювання перед початком моделювання.

Сопрограма довідується, що обробка події завершена, коли вона запитує іншу подію. Функції, які запитують події з ядра моделювання receive() і wait(), так що десь у їхній реалізації втримується виклик transferto(main). Їхній код реалізації в OMNeT++:

```
receive(){
   transferTo(main)
   retrieve current event
   return the event // remember: events = messages
}
wait(){
   create event e
   schedule it at (current sim. time + wait interval)
   transferTo(main)
   retrieve current event
   if (current event is not e) {
```

```
        error
    }
    delete e  // note: actual impl. reuses events
    return
}
```

Таким чином, виклики функцій receive() і wait() – спеціальні точки у функції activity(), тому що вони перебувають там, де час моделювання повинний протікати в модулі; інші модулі одержують можливість почати виконання.

Модулі, написані з activity() потребують стартових повідомлень для запуску. Ці стартові повідомлення вставляються в FES автоматично OMNeT++ на початку моделювання, ще до викликів функцій initialize().

Розроблювач моделі повинен визначити розмір стека процесу для сопрограм вручну. Як правило, досить 16 або 32 кілобайта, але, якщо модуль використовує рекурсивні функції або містить локальні змінні, які займають багато простори в стеці, може знадобитися більше пам'яті. В OMNeT++ є вбудований механізм, який визначає, коли стек модуля занадто малий і переповняється. Omnet ++ також може обчислити обсяг пам'яті стека, який насправді використовує модуль.

Оскільки локальні змінні activity() зберігаються під час подій, у них можна зберігати всю інформацію про стан, буфери пакетів і т.д. Локальні змінні ініціалізуються у верхній частині функції activity(), так що немає особою необхідності у використанні initialize().

Якщо необхідно обробити статистику наприкінці моделювання, то потрібно викликати функцію finish(). Тому що finish() не може одержати доступ до локальних змінних activity(), потрібно помістити змінні й об'єкти, що містять статистичні дані, у клас модуля. Викликати initialize() не потрібно, тому що члени класу також можуть бути ініціалізовані у верхній частині activity(). Таким чином, стандартна установка в псевдокоді виглядає таким чином:

```
class MySimpleModule...{
    ...
    variables for statistics collection
    activity();
    finish();
};
MySimpleModule::activity(){
    declare local vars and initialize them
```

```
    initialize statistics collection variables
    while(true)  {
        ...
    }
}
MySimpleModule::finish(){
    record statistics into file
}
```

Плюси використання activity():

– не потрібен виклик initialize(), стан може зберігатися в локальних змінних activity();

– опис стилю процесу програмується в моделі природно.

Мінуси використання activity():

– обмежена масштабованість: сек сопрограми може неприпустимо перевищити необхідний обсяг пам'яті програми моделювання якщо використовується кілька тисяч або десятків тисяч простих модулів;

– накладні витрати під час виконання: перемикання між сопрограмами виконується трохи повільніше, чим простий виклик функції;

– не забезпечує гарний стиль програмування: використання activity() має тенденцію приводити до ненадійного, заплутаному коду

5.5 Доступ до параметрів модуля

Параметри модуля, оголошені в NED файлі, представлені під час виконання в класі cpar, і доступні шляхом виклику par(), методу cComponent :

```
cPar&delayPar = par("delay");
```

Значення з cPar зчитуються за допомогою методів, які відповідають типам параметрів: boolvalue(), longvalue(), doublevalue(), stringvalue(), stdstringvalue(), xmlvalue().

```
long numjobs = par("numjobs").longvalue();
double processingdelay = par("processingdelay"); // using operator double()
```

Параметр може бути оголошений з модифікатором volatile в NED файлі, він указує на те, що значення параметра перераховується щораз, коли програмі буде потрібно значення під час моделювання. Параметри з volatile, як правило, використовуються для таких речей, як генерація випадкових інтервалів між

пакетами, наприклад, якщо привласнюється значення exponential(1.0) (числа беруться з експоненційого розподілу із середнім значення рівним 1.0).

На противагу цьому, NED параметри без volatile є константами, і зчитування їх значення кілька раз гарантовано дає те ж саме значення. Коли параметру без volatile задається випадкове значення типу exponential(1.0), те воно обчислюється один раз на початку моделювання й заміняється обчисленим результатом, так що всі операції читання одержать одну й те ж значення випадкової величини, обчислене випадковим образом.

Типове використання параметрів без volatile полягає в тому, щоб прочитати їх у методі initialize() класу модуля, і зберегти значення в змінних класу для швидкого доступу в майбутньому:

```
class Source : public csimplemodule{
  protected:
    long numjobs;
    virtual void initialize();
    ...
};
void Source::initialize(){
    numjobs = par("numjobs");
...
}
```

Параметри з volatile перераховуються щораз, коли необхідно одержати їхнє значення. Наприклад, параметр, який являє собою випадково сгенерированій інтервал між пакетами, задається таким чином:

```
void Source::handlemessage(cMessage*msg){
...
scmessaget(simtime() + par("interval").doublevalue(), timermsg);
...
}
```

Наступний код переглядає параметр по імені щораз. Такого пошуку можна уникнути, зберігаючи покажчик на параметр об'єкта в змінну класу. Це показано в наступному коді:

```
class Source : public csimplemodule{
     protected:
```

```
        cPar *intervalp;
    virtual void initialize();
    virtual void handlemessage(cMessage*msg);
...
};
voidsourcmessageialize(){
   intervalp = &par("interval");
   ...
}
void Source::handlemessage(cMessage*msg){
...
scmessaget(simtime() + intervalp->doublevalue(), timermsg);
}
```

Ім'я й тип параметра повертаються методами getname() і gettype() відповідно. Останній повертає значення з безлічі, яка може бути перетворена в рядок, що читається, методом gettypename(). Значення цієї змінної BOOL, DOUBLE, LONG, STRING і XML є внутрішніми типами й, як правило, повинні бути зрозумілі cPar.

Метод isvolatile() указує, чи оголошений параметр в NED файлі з модифікатором volatile. Метод isnumeric() повертає істину, якщо тип параметра double або long. Метод str() повертає значення параметра у вигляді рядка або строкового уявлення виразу. Приклад використання описаних вище методів:

```
int n = getnumparams();
for (int i = 0; i < n; i++){
   cPar& p = par(i);
   EV << "parameter: " << p.getname() << "\n";
   EV << "  type:" <<cPar::getTypeName(p.getType())<<"\n";
EV<<"cparains:"<<p.str()<<"\n";
}
```

NED властивості параметра доступні методом getproperties(), який повертає покажчик на об'єкт cProperties, який зберігає властивості цього параметра. Зокрема, getunit() повертає одиницю виміру параметра (@unit властивість в NED).

Модулі можуть одержувати повідомлення про те, що значення їх параметра змінилося під час виконання, через можливість іншого модуля динамічно

змінювати його. Стандартна дія в такому випадку полягає в перерахуванні зміненого параметра й відновленні стану модуля.

Щоб включити таке повідомлення необхідно перевизначити метод handleparameterchange() класу модуля. Цей метод є зворотним методом (call back), який викликається ядром моделювання у випадку зміни параметрів модуля, але не викликається під час ініціалізації даного модуля.

Оголошення методу виглядає таким чином:

```
void handleparameterchange(const char *parametername);
```

У наступному прикладі показано модуль, який повторно зчитує параметр servicetime, коли його значення змінюється:

```
void Queue::handleparameterchange(const char *parname){
   if (strcmp(parname, "servicetime")==0)
      servicetime = par("servicetime"); // refresh data member
}
```

Щоб одержувати повідомлення під час ініціалізації або завершення, потрібно явно викликати handleparameterchange() з функцій initialize() або finish():

```
for (int i = 0; i < getnumparams(); i++)
   handleparameterchange(par(i).getname());
```

5.6 Доступ до воріт і з’єднань

Ворота (Gates) модуля представлені об'єктами класу cgate. Вони знають, з якими іншими воротами вони зв'язані і якими об'єктами каналів (Channels)Клас cModule має ряд функцій-членів, які працюють із воротами. Можна звернутися до воріт по імені, використовуючи метод gate():

```
cGate *outGate = gate("out");
```

Це працює для вхідних (in) і вихідних (out) воріт. Якщо ворота оголошені в NED як двунаправлені (inout), то, насправді, ядро моделювання представляє їх у вигляді двох воріт, а вищевказаний виклик функції gate() приведе до помилки. Для коректної роботи їй необхідно повідомити, доступ буде до входу

або виходу воріт типу inout. Це виконується шляхом додавання "$i" або "$o" після імені воріт. У наступному прикладі витягуються двоє воріт для воріт g типу inout :

```
cGate *gIn = gate("g$i");
cGate *gOut = gate("g$o");
```

Інший спосіб полягає у використанні функції gatehalf(), яка ухвалює в якості параметрів ім'я воріт і константу Gate::INPUT або cgate::OUTPUT:

```
cGate *gIn = gateHalf("g", cGate::INPUT);
cGate *gOut = gateHalf("g", cGate::OUTPUT);
```

Для перевірки існування конкретних воріт у модуля використовується метод hasgate().

Приклад:

```
if (hasgate("optout"))
   send(newcmessagetOut");
```

Ворота можуть бути також ідентифіковані по числовому коду воріт (ID). Одержати ідентифікатор воріт можна викликом вбудованого методу getid() або методу findgate() модуля, до якого вони включені, указавши ім'я воріт. Метод gate() має перевантажений варіант, який повертає ім'я воріт для заданого ID.

```
int gateid = gate("in")->getid();  // or:
int gateid = findgate("in");
```

Вектори воріт зберігають по одному об'єкту типу cgate у кожному елементі. Щоб одержати доступ до окремих воріт у векторі, необхідно викликати функцію gate() з додатковим параметром – індексом. Номер індексу лежить у межах від 0 до розміру вектора мінус один. Розмір вектора видає метод gatesize(). Вектор воріт не може мати пропуски елементів. Наступний приклад перебирає всі елементи у векторі воріт:

```
for (inti = 0; i<gatesize("out"); i++) {
     cGate *gate = gate("out", i);
}
```

Для воріт типу inout, метод gatesize() викликається з або без суфіксів "$i"/"$o" і повертає той же номер.

Якщо метод hasgate() використовується без індексу, то він повідомляє про існування вектора воріт із заданим іменем незалежно від його розміру. Якщо метод hasgate() використовується з індексом, він повідомляє чи перебуває індекс у межах границь

Доступ до воріт також можна одержати за їх ID. Важливою властивістю ідентифікаторів воріт є те, що їх номер ідентифікатора зростає у відповідності до індексу вектора воріт, тобто ідентифікатор воріт g[k] може бути обчислений як ідентифікатор g[0] плюс k. Цей факт дозволяє ефективно одержувати доступ до будь-яких воріт з вектора воріт, тому що доступ до воріт за допомогою ID більш ефективний, ніж по імені й індексу. Індекс перших воріт може бути отриманий командою gate("out",0)->getid(), але краще використовувати спеціальний метод gatebaseid(), тому що він працює у випадку, коли розмір вектора воріт дорівнює нулю.

Ідентифікатори воріт стабільні й унікальні в межах модуля, тобто ідентифікатор воріт ніколи не змінюється, і в будь-який момент часу ніякі двоє воріт не будуть мати однакові ідентифікатори. Ідентифікатори вилучених воріт не використовуються повторно і є унікальними в житті прогону моделювання.

У наступному прикладі перебирається вектор воріт з використанням ідентифікаторів:

```
int baseid = gatebaseid("out");
int size = gatesize("out");
for (int i = 0; i < size; i++) {
     cGate *gate = gate(baseId + i);.
}
```

Якщо необхідно обійти всі ворота в рамках модуля, то для цього існує два способи. Перший спосіб полягає у виклику методу getgatenames(), який повертає імена всіх воріт і векторів воріт модуля. Потім можна викликати метод isgatevector(name), щоб визначити, чи є конкретне ім'я вектором або скаляром воріт. Після чого вектори воріт необхідно обійти по їхніх індексах. Для воріт типу inout метод getgatenames() повертає базове ім'я воріт без суфіксів "$i"/"$o", тому ці два напрямки повинні оброблятися окремо. Метод gatetype(name) перевіряє, чи мають ворота тип inout, input або output (він повертає константи cGate::INOUT, cGate::INPUT, або cGate::OUTPUT).

Другий спосіб полягає у використанні класу Gateiterator наданий класом cmodule. Цей спосіб виглядає таким чином:

```
for (cModule::GateIteratori(this);!i.end();i++){
```

```
cGate *gate = i();
...
}
```

Тут змінна this указує на модуль, ворота якого перелічуються в циклі (її можна замінити на будь-яку змінну типу *cmodule).

Іноді потрібно додавати або видаляти ворота під час виконання моделювання. Можна додавати скалярні ворота й вектори воріт, змінювати розмір векторів воріт і видаляти скалярні ворота й цілі вектори воріт. У свою чергу не можна вилучити окремо обрані ворота з вектора воріт, вилучити одну половинку воріт типу inout (наприклад, "gate$o") або встановити різні розміри двом половинам вектора воріт типу inout.

Методи класу cmodule для додавання й видалення воріт відповідно addgate(name,type,isvector=false) і deletegate(name). Розмір вектора воріт можна змінити командою setgatesize(name,size). Жоден із цих методів не приймає суфікси "$i" / "$o" в іменах воріт.

5.7 Висновки

У розділі наведено концепцію моделювання у OMNeT++, яка опирається на технологію календарного імітаційного моделювання. Загальні розрахунки виконуються у простих модулях, які, у свою чергу написані мовою C++.

У розділі розповідається про пул майбутніх повідомлень, загальні принципи вибору повідомлень з нього для обробки. Принципи та переваги використання методи initialize() і finish() обґрунтовано для різних ситуацій.

У розділі порівнюються функції handlemessage() та activity(). Визначено плюси та мінуси їх використання.

Контрольні питання.

1. Концепції моделювання у OMNeT++.
2. Дискретна система подій. Обробка події.
3. Повідомлення у OMNeT++. Клас cmessage.
4. Правила вибору повідомлення з FES.
5. Час моделювання в Omnet++. Тип даних simtime_t.
6. Клас ccomponent та його функції.
7 Визначення типів простих модулів

8. Методи initialize() і finish().
9. Порядок викликів функцій finish().
10. Додавання функціональності в csimplemodule.
11. Призначення та використання функцій handlemessage() і activity().
12. Перелік функції, які пов'язані з повідомленнями або подіями, які використовуються в handlemessage().
13. Яку інформацію містять фрагменти даних, які додаються до класу модуля?
14. У яких випадках handlemessage() переважніше використовувати, замість activity()?
15. Плюси та мінуси використання handlemessage().
16. Які найбільш важливі функції використовуються з activity()?
17. Яким чином виконується доступ до параметрів модуля?
18. Яким чином виконується доступ до воріт і з’єднань?

ВИСНОВКИ

Хоча Omnet++ сам по собі не є мережним симулятором, він одержав широку популярність у користувачів у науковому співтоваристві й промислових підприємствах у якості мережної платформи для моделювання.

Omnet++ надає архітектуру компонентів для моделей. Компоненти (модулі) розроблені на C++, а потім зібрані в більші компоненти й моделі з використанням мови високого рівня (NED). Можливість повторного використання моделей поставляється безкоштовно. Omnet++ має широку підтримку графічного інтерфейсу, і, завдяки своїй модульній архітектурі, ядро моделювання й моделі можуть бути легко вбудовані в розроблювальні користувачами додатки.

Вузли мережі з'єднуються між собою каналами зв'язку, параметри котрих можна змінювати. На основному вікні візуалізації відображається внутрішня структура сервера і маршрутизатора (протоколи мережевого, транспортного рівнів моделі OSI, таблиця маршрутизації і вказано запис логу всіх процесів, які відбуваються. Всі вищезгадані властивості доступні в будь-який час протікання процесу. Тим самим, користувач може відстежити пересування пакета, який його цікавить, по всій мережі.

До системи OMNeT++ закладено детальну реалізацію 24-х протоколів передачі різних рівнів моделі ISO-OSI, при чому є можливість написання і підключення власних додаткових модулів та підмодулів. У системі присутній розвинений графічний інтерфейс.

Список використаних джерел

1. *Хемди А. Таха* Введение в исследование операций. 7-е издание / Хемди А. Таха – М.: Издательский дом «Вильямс», 2005. – 912с.

2. *Мину М.* Математическое программирова-ние / М. Мину – М.: Наука, 1990. – 488с.

3. *Карманов В.Г.* Математическое программирование / В.Г. Карманов – М.: Наука, 1980. – 256с.

4. *Хедли Дж.* Нелинейное и динамическое программирование / Дж. Хедли – М.: Мир, 1967. – 561с.

5. *Аттетков А.В.* Методы оптимизации. – А.В. Аттетков, С.В. Галкин, В.С.Зарубин – М.: Изд. МГТУ им. Баумана, 2003. – 439с.

6. Логистика: Учебное пособие / под. ред. Б.А. Аникина. – М.: ИНФРА-М, 1999. – 327с.

7. Практикум по логистике: Учебное пособие. – 2е изд. / под. ред. Б.А. Аникина. – М.: ИНФРА-М, 2006. – 276с.

8. Корпоративная логистика. 300 ответов на вопросы профессионалов / под редакцией В.И. Сергеева – М.: ИНФРА-М, 2005. – 976с.

9. *Мешкова Л.Л.* Логистика в сфере материальных услуг. 2-е изд. / Л.Л. Мешкова, И.И. Белоус, Н.М. Фролов – Тамбов: Изд-во Тамб. гос. техн. ун-та, 2002. – 188с.

10. Основы логистики: Учебное пособие / под ред. Л.Б. Миротина и В.И. Сергеева. – М.: ИНФРА-М, 2000. – 200с

11. *Неруш Ю.М.* Логистика: учебное пособие. 4-е изд. / Ю.М. Неруш – М.: Изд-во Проспект, 2006. – 520с

12. *Сергеев В.И.* Логистика: Информационные системы и технологии: Учебно-практическое пособие. / В.И. Сергеев, М.Н. Григорьев, С.А. Уваров – М.: Издательство «Альфа-Пресс», 2008. – 608с.

13. *Кобзарь А.И.* Прикладная математическая статистика / А.И. Кобзарь — М.: Физматлит, 2006. — 816 с.