1. GIỚI THIỆU

1.1. OMNeT++ là gì?

OMNeT++ là viết tắt của cụm từ Objective Modular Network Testbed in C++.

OMNeT++ là một ứng dụng cung cấp cho người sử dụng môi trường để tiến hành mô phỏng hoạt động của mạng. Mục đích chính của ứng dụng là mô phỏng hoạt động mạng thông tin, tuy nhiên do tính phổ cập và linh hoạt của nó, OMNeT++ còn được sử dụng trong nhiều lĩnh vực khác như mô phỏng các hệ thống thông tin phức tạp, các mạng kiểu hàng đợi (queueing networks) hay các kiến trúc phần cứng...

OMNeT++ cung cấp sẵn các thành phần tương ứng với các mô hình thực tế. Các thành phần này (còn được gọi là các module) được lập trình theo ngôn ngữ C++, sau đó được tập hợp lại thành những thành phần hay những mô hình lớn hơn bằng một ngôn ngữ bậc cao (NED). OMNeT++ hỗ trợ giao diện đồ hoạ, tương ứng với các mô hình cấu trúc của nó đồng thời phần nhân mô phỏng (simulation kernel) và các module của OMNeT++ cũng rất dễ dàng nhúng vào trong các ứng dụng khác.

1.2. Các thành phần chính của OMNeT++

- Thư viện phần nhân mô phỏng (simulation kernel)
- Trình biên dịch cho ngôn ngữ mô tả hình trạng (topology description language) - NED (nedc)
- Trình biên tập đồ hoạ (graphical network editor) cho các file NED (GNED)
- Giao diện đồ hoạ thực hiện mô phỏng, các liên kết bên trong các file thực hiện mô phỏng (Tkenv)
- Giao diện dòng lệnh thực hiện mô phỏng (Cmdenv)
- Công cụ (giao diện đồ hoạ) vẽ đồ thị kết quả vector ở đầu ra (Plove)
- Công cụ (giao diện đồ hoạ) mô tả kết quả vô hướng ở đầu ra (Scalars)
- Công cụ tài liệu hoá các mô hình
- Các tiện ích khác
- Các tài liệu hướng dẫn, các ví dụ mô phỏng...

1.3. Ứng dụng

OMNeT++ là một công cụ mô phỏng các hoạt động mạng bằng các module được thiết kế hướng đối tượng. OMNeT++ thường được sử dụng trong các ứng dụng chủ yếu như:

- Mô hình hoạt động của các mạng thông tin

- Mô hình giao thức
- Mô hình hoá các mạng kiểu hàng đợi
- Mô hình hoá các hệ thống đa bộ vi xử lý (multiprocesser) hoặc các hệ thống phần cứng theo mô hình phân tán khác (distributed hardware systems)
- Đánh giá kiến trúc phần cứng
- Đánh giá hiệu quả hoạt động của các hệ thống phức tạp...

1.4. Mô hình trong OMNeT++

Một mô hình trong OMNeT++ bao gồm các module lồng nhau có cấu trúc phân cấp. Độ sâu của của các module lồng nhau là không giới hạn, điều này cho phép người sử dụng có thể biểu diễn các cấu trúc logic của các hệ thống trong thực tế bằng các cấu trúc mô hình. Các module trao đổi thông tin với nhau thông qua việc gửi các message (message). Các message này có thể có cấu trúc phức tạp tuỳ ý. Các module có thể gửi các message này theo hai cách, một là gửi trực tiếp tới địa chỉ nhận, hai là gửi đi theo một đường dẫn được định sẵn, thông qua các cổng và các kết nối.

Các module có thể có các tham số của riêng nó. Các tham số này có thể được sử dụng để chỉnh sửa các thuộc tính của module và để biểu diễn cho topology của mô hình.

Các module ở mức thấp nhất trong cấu trúc phân cấp đóng gói các thuộc tính. Các module này được coi là các module đơn giản, và chúng được lập trình trong ngôn ngữ C++ bằng cách sử dụng các thư viện mô phỏng.

2. TỔNG QUAN

2.1. Khái niệm mô hình hoá

OMNeT++ cung cấp cho người sử dụng những công cụ hiệu quả để mô tả cấu trúc của các hệ thống thực tế.

Các module lồng nhau có cấu trúc phân cấp

Các module là các đối tượng cụ thể của các kiểu module

Các module trao đổi thông tin bằng các message qua các kênh

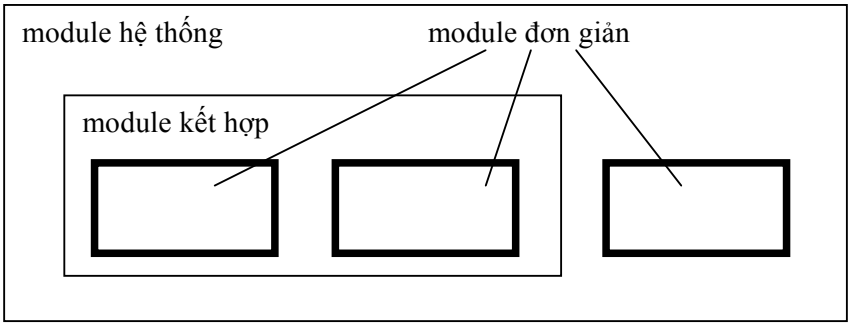
Các tham số của module linh hoạt

Ngôn ngữ mô tả topology

2.1.1. Cấu trúc phân cấp của các module

Một mô hình trong OMNeT++ chứa các module lồng nhau có cấu trúc phân cấp, trao đổi thông tin với nhau bằng cách gửi các message. Mỗi mô hình này thường biểu diễn cho một hệ thống mạng. Module mức cao nhất trong cấu trúc phân cấp được gọi là module hệ thống. Module này có thể chứa các module con, các module con cũng có thể chứa các module con của riêng nó. Độ sâu phân cấp đối với các module là không giới hạn, điều này cho phép người sử dụng có thể dễ dàng biểu diễn một cấu trúc logic của một hệ thống trong thực tế bằng cấu trúc phân cấp của OMNeT++.

Cấu trúc của mô hình có thể được mô tả bằng ngôn ngữ NED của OMNeT++



Hình I-2.1 - Các module đơn giản và kết hợp

Các module có thể chứa nhiều module con và được gọi là module kết hợp. Các module đơn giản là các module có cấp thấp nhất trong cấu trúc phân cấp. Các module đơn giản chứa các thuật toán của mô hình. Người sử dụng triển khai các module đơn giản bằng ngôn ngữ C++, sử dụng các thư viện mô phỏng của OMNeT++.

2.1.2. Kiểu module

Tất cả các module dù là đơn giản hay phức tạp đều là các đối tượng cụ thể của các kiểu module. Trong khi mô tả các mô hình, người sử dụng định nghĩa ra các kiểu

module; các đối tượng cụ thể của các kiểu module này được sử dụng như các thành phần của các kiểu module phức tạp hơn. Cuối cùng, người sử dụng tạo module hệ thống như một đối tượng cụ thể của kiểu module đã được định nghĩa trước đó, tất cả các module của mạng đều là module con (hoặc là con của module con) của module hệ thống.

Khi một kiểu module được sử dụng như một khối dựng sẵn (building block), sẽ không thể phân biệt đó là một module đơn giản hay phức tạp. Điều này cho phép người sử dụng có thể tách các module đơn giản ra thành nhiều module đơn giản được nhúng trong một module kết hợp, và ngược lại có thể tập hợp các chức năng của một module kết hợp trong một module đơn giản mà không ảnh hưởng gì đến các kiểu module đã được người sử dụng định nghĩa.

Kiểu module có thể được lưu trữ trong một file riêng rẽ. Điều này cho phép người sử dụng có thể nhóm các kiểu module lại và tạo ra một thư viện thành phần

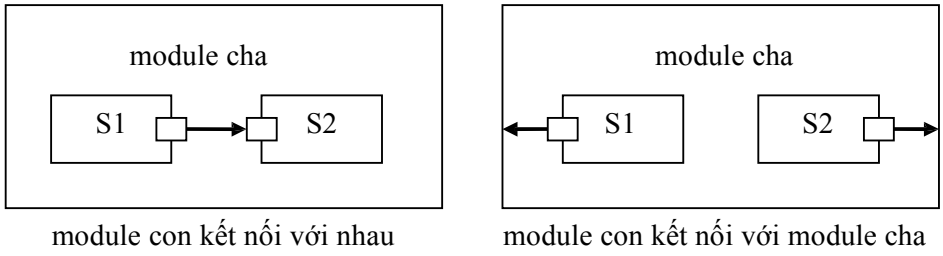
2.1.3. Message, cổng, liên kết

Các module trao đổi thông tin bằng việc gửi các message. Trong thực tế, message có dạng khung (frame) hoặc là các gói tin (packet) được truyền đi trong mạng. Các message có thể có cấu trúc phức tạp tuỳ ý. Các module đơn giản có thể gửi các message đi một cách trực tiếp đến vị trí nhận hoặc gửi đi theo một đường dẫn định sẵn thông qua các cổng và các liên kết.

"Thời gian mô phỏng địa phương" (local simulation time) của một module tăng lên khi module nhận được một message. Message có thể đến từ một module khác hoặc đến từ cùng một module (message của chính bản thân module - self-message được dùng để thực hiện bộ định thời).

Cổng (gate) là các giao tiếp vào ra của module. Message được gửi đi qua các cổng ra và được nhận vào thông qua các cổng vào.

Mỗi kết nối (connection) hay còn gọi là liên kết (link) được tạo bên trong một mức đơn trong cấu trúc phân cấp của các module: bên trong một module kết hợp, một kết nối có thể được tạo ra giữa các cổng tương ứng của hai module con, hoặc giữa cổng của module con với cổng của module kết hợp.



Hình I-2.2 - Các kết nối

Tương ứng với cấu trúc phân cấp của một mô hình, các message thường di chuyển qua một loạt các kết nối với điểm bắt đầu và kết thúc là các module đơn giản. Tập các kết nối đi từ một module đơn giản và đến một module đơn giản được gọi là route. Các module kết hợp hoạt động giống như các "cardboard box" trong mô hình, "trong suốt" trong việc chuyển tiếp các message giữa các thành phần bên trong và thế giới bên ngoài.

2.1.4. Mô hình truyền gói tin

Một kết nối có thể có ba tham số đặc trưng. Những tham số này rất thuận tiện cho các mô hình mô phỏng mạng thông tin nhưng không hữu dụng lắm cho các kiểu mô hình khác. Ba tham số này bao gồm:

- Độ trễ đường truyền (propagation delay) tính bằng s - giây.
- Tỉ số lỗi bit, được tính bằng số lỗi/bit.
- Tỉ số dữ liệu, được tính bằng số bit/s.

Các tham số này là tuỳ chọn. Giá trị của các tham số này là khác nhau trên từng kết nối, phụ thuộc vào kiểu của liên kết (hay còn gọi là kiểu của kênh truyền - channel type).

Độ trễ đường truyền là tổng thời gian đến của message bị trễ đi khi truyền qua kênh.

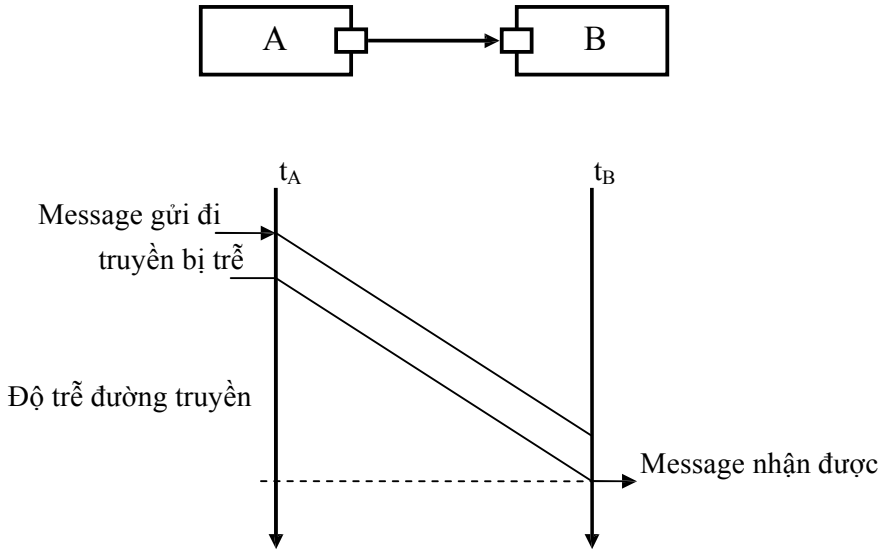
Tỉ số lỗi bit ảnh hưởng đến quá trình truyền message qua kênh. Tỉ số này là xác suất các bit bị truyền sai. Do đó xác suất để một message độ dài n bit truyền đi chính xác là:

P(message gửi đi được nhận chính xác) = (1 - ber)n

trong đó ber là tỉ số lỗi bit và n là số bit của message.

Các message truyền đi đều có một cờ lỗi, cờ này sẽ được thiết lập khi việc truyền message có lỗi.

Tỉ số dữ liệu được tính theo đơn vị bit/s, và nó được sử dụng để tính thời gian để truyền một gói tin. Khi tỉ số này được sử dụng, quá trình gửi message đi trong mô hình sẽ tương ứng với việc truyền bit đầu tiên và message được tính là đến nơi sau khi bên nhận đã nhận được bit cuối cùng.



Hình I-2.3 - Truyền message

2.1.5. Tham số

Các module có thể các tham số.Các tham số này có thể được đặt giá trị trong các file NED hoặc các file cấu hình ompnetpp.ini.

Các tham số này có thể được dùng để thay đổi các thuộc tính của các module đơn giản hoặc dùng để biểu diễn cho topology của mô hình.

Các tham số có thể có kiểu là chuỗi, số học, giá trị logic hoặc cũng có thể chứa cây dữ liệu XML (XML data tree). Các biến kiểu số trong các biểu thức có thể nhận giá trị từ các tham số khác, gọi hàm, sử dụng các biến ngẫu nhiên từ các nguồn phân tán hoặc nhận giá trị trực tiếp được nhập vào bởi người sử dụng.

Các tham số có kiểu số có thể được dùng để cấu hình topology rất dễ dàng. Nằm trong các module kết hợp, các tham số này có thể được dùng để chỉ ra số module con, số cổng giao tiếp và cách các kết nối nội bộ được tạo ra.

2.1.6. Phương pháp mô tả topology

Người sử dụng dùng ngôn ngữ NED (Network Description) để mô tả cấu trúc của các mô hình

---

2.2. Lập trình thuật toán

Các module đơn giản có thể chứa các thuật toán như các hàm của C++. Sự linh hoạt và sức mạnh của C++, kết hợp với các thư viện mô phỏng của OMNeT++ tạo điều kiện dễ dàng cho người sử dụng. Các lập trình viên mô phỏng có thể chọn lựa việc mô tả theo sự kiện hay theo tiến trình, có thể dễ dàng sử dụng những khái niệm của lập trình hướng đối tượng (như đa hình, kế thừa) và thiết kế các mẫu thử (pattern) để mở rộng chức năng của quá trình mô phỏng.

Các đối tượng mô phỏng (message, module, queue...) được thể hiện qua các lớp của C++. Một số lớp cơ bản trong thư viện mô phỏng của OMNeT++:

- Module, cổng, liên kết...
- Các tham số
- Message
- Các lớp Container (mảng, hàng đợi...)
- Các lớp Data Collection

Các lớp này có thể được sử dụng như những công cụ cho phép người sử dụng có thể duyệt qua tất cả các đối tượng khi chạy thử mô hình đồng thời hiển thị thông tin về chúng như tên của đối tượng, tên lớp, các biến trạng thái và nội dung bên trong. Đặc điểm này cũng cho phép tạo ra các mô hình mô phỏng có giao diện đồ hoạ (GUI) với phần cấu trúc bên trong được che đi.

2.3. Sử dụng OMNeT++

2.3.1. Xây dựng và chạy thử các mô hình mô phỏng

Một mô hình OMNeT++ bao gồm những phần sau:

- Ngôn ngữ mô tả topology - NED (file có phần mở rộng .ned): mô tả cấu trúc của module với các tham số, các cổng... Các file .ned có thể được viết bằng bất kỳ bộ soạn thảo hoặc sử dụng chương trình GNED có trong OMNeT++.
- Định nghĩa cấu trúc của các message (các file có phần mở rộng .msg): Người sử dụng có thể định nghĩa rất nhiều kiểu messsage và thêm các trường dữ liệu cho chúng. OMNeT++ sẽ dịch những định nghĩa này sang các lớp C++ đầy đủ.
- Mã nguồn của các module đơn giản. Đây là các file C++ với phần mở rộng là .h hoặc .cc.

Hệ thống mô phỏng cung cấp cho ta các thành phần sau:

- Phần nhân mô phỏng. Phần này chứa code để quản lý quá trình mô phỏng và các thư viện lớp mô phỏng. Nó được viết bằng C++, được biên dịch và được đặt cùng dạng với các file thư viện (các file có phần mở rộng là .a hoặc .lib).
- Giao diện người sử dụng. Giao diện này được sử dụng khi thực hiện quá trình mô phỏng, tạo sự dễ dàng cho quá trình sửa lỗi, biểu diễn (demonstration) hoặc khi thực hiện mô phỏng theo từng khối (batch execution of simulations). Có một vài kiểu giao diện trong OMNeT++, tất cả đều được viết bằng C++, được biên dịch và đặt cùng nhau trong các thư viện (các file có phần mở rộng là .a hoặc .lib).

Thực hiện mô phỏng và phân tích kết quả

Các chương trình thực hiện mô phỏng (the simulation executable) là các chương trình độc lập, tức là nó có thể chạy trên các máy khác không cài đặt OMNeT++ hay các file mô hình tương ứng. Khi chương trình khởi động, nó bắt đầu đọc file cấu hình (thông thường là file omnetpp.ini). File này chứa các thiết lập để điều khiển quá trình mô phỏng thực hiện, các biến cho các tham số của mô hình... File cấu hình cũng có thể được sử dụng để điều khiển nhiều quá trình mô phỏng, trong trường hợp đơn giản nhất là các quá trình mô phỏng này sẽ được thực hiện lần lượt bởi một chương trình mô phỏng (simulation program).

Đầu ra của quá trình mô phỏng là các file dữ liệu. Các file này có thể là các file vector, các file vô hướng hoặc các file của người sử dụng. OMNeT++ cung cấp một công cụ đồ hoạ Plove để xem và vẽ ra nội dung của các file vector. Tuy nhiên chúng ta cũng nên hiểu rằng khó mà có thể xử lý đầy đủ các file kết quả mà chỉ dùng riêng OMNeT++; các file này đều là các file có định dạng để có thể đọc được bởi các gói xử lý toán học của các chương trình như Matlab hay Octave, hoặc có thể được đưa vào bảng tính của các chương trình như OpenOffice Calc, Gnumeric hay Microsoft Excel. Tất cả các chương trình này đều có chức năng chuyên dụng trong việc phân tích số hoá, vẽ biểu diễn (visualization) vượt qua khả năng của OMNeT++.

Các file vô hướng cũng có thể được biểu diễn bằng công cụ Scalar. Nó có thể vẽ được các biểu đồ, các đồ thị dựa vào tập hợp các toạ độ (x, y) và có thể xuất dữ liệu vào clipboard để có thể sử dụng trong các chương trình khác nhằm đưa những phân tích chi tiết hơn.

Giao diện người sử dụng

Mục đích chính của giao diện người sử dụng là che những phần phức tạp bên trong cấu trúc của các mô hình đối với người sử dụng, dễ dàng điều khiển quá trình mô phỏng, và cho phép người sử dụng có khả năng thay đổi các biến hay các đối tượng bên trong của mô hình. Điều này là rất quan trọng đối với pha phát triển và sửa lỗi trong dự án. Giao diện đồ hoạ cũng có thể được sử dụng để trình diễn hoạt động của mô hình.

Cùng một mô hình người sử dụng có thể trên nhiều giao diện khác nhau mà không cần phải thay đổi gì trong các file mô hình. Người sử dụng có thể kiểm thử và sửa lỗi rất dễ dàng qua giao diện đồ hoạ, cuối cùng có thể chạy nó dựa trên một giao diện đơn giản và nhanh chóng có hỗ trợ thực hiện theo khối (batch execution).

Các thư viện thành phần

Các kiểu module có thể được lưu tại những vị trí độc lập với chỗ mà chúng thực sự được sử dụng. Đặc điểm này cung cấp cho người sử dụng khả nhóm các kiểu module lại với nhau và tạo ra các thư viện thành phần.

Các chương trình mô phỏng độc lập

Các chương trình thực hiện quá trình mô phỏng có thể được lưu nhiều lần, không phụ thuộc vào các mô hình, sử dụng cùng một thiết lập cho các module đơn giản. Người sử dụng có thể chỉ ra trong file cấu hình mô hình nào sẽ được chạy. Điều này tạo khả năng cho người sử dụng có thể xây dựng những chương trình thực hiện lớn bao gồm nhiều quá trình mô phỏng, và phân phối nó như một công cụ mô phỏng độc lập. Khả năng linh hoạt của ngôn ngữ mô tả topology cũng hỗ trợ cho hướng tiếp cận này.

2.3.2. Hệ thống file

Sau khi cài đặt OMNet++, thư mục omnetpp trên hệ thống máy của bạn nên chứa các thư mục con dưới đây.

Hệ thống mô phỏng:

omnetpp/	thư mục gốc của OMNeT++
bin/	các công cụ trong OMNeT++ (GNED, nedtool...)
include/	các file header cho mô hình mô phỏng
lib/	các file thư viện
bitmaps/	các biểu tượng đồ hoạ
doc/	các file hướng dẫn, readme...
manual/	file hướng dẫn dạng HTML
tictoc-tutorial/	giới thiệu sử dụng OMNeT++
api/	API tham chiếu dạng HTML
nedxml-api/	API tham chiếu cho thư viện NEDXML
src/	mã nguồn của tài liệu
src/	mã nguồn của OMNeT++
nedc/	nedtool, trình biên dịch message
sim/	phần nhân mô phỏng
parsim/	các file dành cho việc thực hiện phân tán
netbuilder/	các file dành cho việc đọc động các file NED
envir/	mã nguồn cho giao diện người sử dụng
cmdenv/	giao diện người dùng dòng lệnh

	tkenv/	giao diện người sử dụng dựa trên Tcl/tk
	gned/	công cụ soạn thảo file NED
	plove/	công cụ vẽ và phân tích đầu ra dạng vector
	scalars/	công cụ vẽ và phân tích đầu ra dạng vô hướng
	nedxml/	thư viện NEDXML
	utils/	các tiện ích khác...
test/		bộ kiểm thử lùi
	core/	bộ kiểm thử lùi cho thư viện mô phỏng
	distrib/	bộ kiểm thử lùi
	...	

Các quá trình mô phỏng mẫu được chứa trong thư mục samples

samples/		thư mục chứa các mô hình mô phỏng mẫu
	aloha/	mô hình của giao thức Aloha
	cqn/	Closed Queue Network
	...	

Thư mục contrib chứa các chương trình có thể kết hợp với OMNeT++

contrib/		
	octave/	script của Octave dùng để xử lý kết quả
	emacs/	bộ đánh dấu cú pháp NED cho Emacs

Ngoài ra bạn cũng có thể tìm thấy các thư mục khác như msvc/, chứa các thành phần tích hợp cho Microsoft Visual C++...

3. NGÔN NGỮ NED

3.1 Tổng quan về NED

NED được sử dụng để mô tả topology của một mô hình trong OMNeT++. NED sử dụng phương pháp mô tả module hoá. Điều này có nghĩa là một mạng có thể được mô tả như một tập hợp các mô tả thành phần (các kênh, các kiểu module đơn giản hay kết hợp). Các kênh, các kiểu module đơn giản và kết hợp được sử dụng để mô tả một mạng nào đó có thể được sử dụng lại khi mô tả một mạng khác.

Các file chứa mô tả mạng thường có phần mở rộng là .ned. Các file NED có thể được load động vào các chương trình mô phỏng, hay có thể được dịch sang C++ bằng bộ biên dịch của NED và được liên kết bên trong các chương trình thực hiện.

3.1.1. Các thành phần của ngôn ngữ mô tả NED

Một filel NED bao gồm các phần như sau:

- Các chỉ dẫn import
- Khai báo các kênh
- Khai báo các module đơn giản và kết hợp
- Khai báo mạng

3.1.2. Các từ khoá

Người sử dụng cần phải chú ý không sử dụng những từ khoá có sẵn của NED để đặt tên cho các đối tượng khác. Các từ khoá cơ bản của NED bao gồm:

import channel endchannel simple endsimple module endmodule error delay datarate const parameters gates submodules connections atesizes if for do endfor network endnetwork nocheck ref ancestor true false like input numeric string bool char xml xmldoc

3.1.3. Đặt tên

Trong NED người sử dụng có thể đặt tên cho các module, các kênh, các module con, các tham số, các cổng, các thuộc tính và hàm chức năng của kênh... Các tên này có thể bao gồm các chữ cái tiếng Anh, các chữ số và dấu gạch dưới "_". Tên luôn được đặt bắt đầu bằng chữ cái hoặc dấu gạch dưới. Trong trường hợp muốn đặt tên bắt đầu bằng chữ số, bạn có thể sử dụng thêm một dấu gạch dưới đặt ở đầu, ví dụ như _3Com...

Nếu tên bao gồm nhiều từ nên viết hoa ở đầu mỗi từ hoặc có thể sử dụng dấu gạch dưới. Tên của các module, kênh và mạng nên bắt đầu bằng chữ cái in hoa còn tên của tham số, cổng và các module con nên bắt đầu bằng chữ cái thường.

NED là một ngôn ngữ có phân biệt hoa thường.

3.1.4. Chú thích

Các dòng chú thích có thể đặt ở bất kì vị trí nào trong file NED. Tương tự như cú pháp của C++, các dòng chú thích trong NED bắt đầu bằng dấu '//'.

Chú thích trong NED có thể được sử dụng trong những công cụ tạo tài liệu (document generator) như JavaDoc, Doxygen

---

3.2. Các chỉ dẫn import

Từ khoá import được sử dụng để thêm các khai báo trong các file mô tả khác. Sau khi đã import, người sử dụng có thể sử dụng tất cả các thành phần đã được định nghĩa trong file mô tả đó.

Chú ý khi thêm một file mô tả, chỉ có các thông tin khai báo được sử dụng. Cũng tương tự như vậy khi một file được thêm vào không có nghĩa là nó sẽ được dịch khi file chứa nó được dịch. Người sử dụng sẽ phải dịch tất cả các file chứ không phải chỉ là file ở mức cao nhất.

Bạn có thể xác định một file thêm vào mà có hoặc không viết phần mở rộng.

Ví dụ:

import "ethenet"; //import ethernet.ned

Bạn cũng có thể sử dụng đường dẫn trong khi sử dụng từ khoá import hoặc tốt hơn là bạn sử dụng trình biên dịch của NED với tham số -I để đặt tên cho thư mục chứa các file mà bạn muốn import.

3.3. Khai báo các kênh

Một định nghĩa kênh được dùng để xác định kiểu kết nối. Tên của kênh có thể được sử dụng sau đó trong file để tạo các liên kết với các tham số khác.

Cú pháp:

channel Tên kênh

//...

endchannel

Ba tham số có thể được gán giá trị trong phần thân của đoạn mã khai báo kênh, tất cả các tham số này đều là các tuỳ chọn: độ trễ, lỗi và tốc độ dữ liệu (datarate). Độ trễ là thời gian trễ trên đường truyền được tính bằng giây. Lỗi là tham số đặc trưng cho xác suất truyền sai một bit trên đường truyền. Tốc độ dữ liệu là tham số được tính bằng độ rộng băng thông của kênh truyền, được tính bằng bit/s và được dùng để tính thời gian truyền của một gói tin. Các thuộc tính có thể xuất hiện theo bất kỳ thứ tự nào trong khai báo.

Giá trị của các tham số (thuộc tính) nên là các hằng số.

Ví dụ:

channel LeasedLine

delay 0.0018 // sec

error 1e-8

datarate 128000 // bit/sec

endchannel

3.4. Khai báo các module đơn giản

Các module đơn giản là các khối chương trình được xây dựng sẵn cho các module khác (có thể là các module kết hợp). Các module được khai báo bằng tên và theo quy ước tên của các module này được đặt tên bắt đầu bằng chữ cái in hoa.

Các module đơn giản được khai báo thông qua các cổng và các tham số.

Cú pháp:

simple SimpleModuleName

parameters:

//...

gates:

//...

endsimple

3.4.1. Các tham số của module đơn giản

Các tham số là các biến phụ thuộc vào từng mô hình. Tham số của các module đơn giản được sử dụng bởi các hàm (hay còn được gọi là các thuật toán của module) khai báo trong chính module. Theo quy ước các tham số sẽ được đặt tên bắt đầu bằng chữ cái thường.

Các tham số được khai báo bằng cách liệt kê tên sau từ khoá parameters. Kiểu của các tham số có thể là kiểu số (numeric), hằng số (numeric const hay viết gọn là const), giá trị logic (bool), kiểu chuỗi (string) hoặc xml. Khi tham số không khai báo rõ kiểu thì mặc định kiểu của tham số đó là numeric.

Ví dụ:

simple TrafficGen

parameters:

interarrivalTime,

numOfMessages : const,

address : string;

gates: //...

endsimple

Các tham số có thể được gán giá trị từ NED (khi các module được sử dụng như các khối dựng sẵn của một khối kết hợp lớn hơn) hoặc từ file cấu hình omnetpp.ini.

Tham số ngẫu nhiên và hằng số

Các tham số có kiểu số có thể được đặt để trả về một giá trị ngẫu nhiên theo dạng phân phối đều (uniformly distributions) hoặc các dạng phân phối khác

Trong đa số trường hợp, các tham số thường chỉ nhận giá trị ngẫu nhiên khi bắt đầu khởi gán, sau đó giá trị này được giữ nguyên. Khi đó các tham số này phải được khai báo là hằng số - const. Khai báo một tham số là const thì giá trị của tham số sẽ được xác định một lần duy nhất khi bắt đầu thực hiện mô phỏng và sau đó giá trị đó sẽ được coi là hằng số. (Chú ý OMNeT++ khuyến khích việc khai báo mọi tham số là const trừ những trường hợp bạn muốn sử dụng chức năng tạo số ngẫu nhiên).

Tham số XML

Đôi khi các module cần đầu vào là những thông tin phức tạp hơn khả năng mô tả của các tham số, khi đó bạn có thể sử dụng một file cấu hình mở rộng. OMNeT++ có thể đọc và xử lý các file này thông qua một tham số chứa tên của file.

Từ các phiên bản 3.0 trở lên, XML được coi là một dạng chuẩn cho các file cấu hình và OMNeT++ cũng tích hợp sẵn các công cụ cho phép người sử dụng có thể làm việc được với các file XML. OMNeT++ chứa bộ phân tích cú pháp XML (XML parser), có khả năng đọc các file DTD, sử dụng bộ nhớ đệm để nhớ các file (trong trường hợp một file XML được tham chiếu tới nhiều module thì nó sẽ cũng chỉ phải load một lần), cung cấp cho người sử dụng khả năng chọn lọc các phần trong tài liệu thông qua XPath, thể hiện nội dung của file XML thông qua DOM.

3.4.2. Các cổng của module đơn giản

Cổng là các điểm kết nối của module. Điểm bắt đầu và kết thúc một kết nối giữa hai module chính là các cổng. OMNeT++ hỗ trợ kiểu kết nối một chiều (đơn công) do đó có hai loại cổng là cổng vào và cổng ra. Các message được gửi đi từ cổng ra và được nhận vào từ cổng vào. Theo quy ước, các cổng được đặt tên bắt đầu bằng chữ cái thường.

Ở đây chúng ta có khái niệm về các vector cổng trong đó một vector cổng là một tập hợp bao gồm nhiều cổng đơn.

Cổng được khai báo bằng cách khai báo tên sau từ khoá gates. Cặp dấu [] thể hiện một vector cổng. Các thành phần của một vector cổng được đánh số bắt đầu từ 0.

Ví dụ:

simple NetworkInterface

parameters: //...

gates:

in: fromPort, fromHigherLayer;

out: toPort, toHigherLayer;

endsimple

simple RoutingUnit

parameters: //...

gates:

in: output[];

out: input[];

endsimple

Kích thước của một vector cổng có thể được xác định sau do đó mỗi đối tượng cụ thể của một mô hình có thể có các vector cổng có kích thước khác nhau.

3.5. Khai báo module kết hợp

Module kết hợp là các module có thể chứa một hoặc nhiều các module con. Bất kỳ kiểu module nào (đơn giản hay kết hợp) đều có thể được dùng như là một module con. Cũng giống như các module đơn giản, các module kết hợp cũng có các cổng, các tham số và chúng có thể được sử dụng ở bất kỳ chỗ nào mà các module đơn giản có thể được sử dụng.

Hình tượng hoá chúng ta có thể tưởng tượng các module kết hợp giống như các hộp bìa cứng mà chúng ta có thể giấu phần mô hình mô phỏng và các cấu trúc phức tạp bên trong nó. Không có các hành vi tích cực (active behaviour) nào liên quan đến các module kết hợp - chúng chỉ đơn giản là một nhóm các module kết hợp trong một thành phần lớn hơn để có thể được sử dụng như một mô hình hoặc như một khối dựng sẵn cho các module kết hợp khác.

Theo quy ước, tên của các module (bao gồm cả kiểu module kết hợp) đều được bắt đầu bằng chữ hoa.

Các module con có thể sử dụng các tham số của module cha. Các module con này có thể kết nối với nhau hoặc/và kết nối với module kết hợp chứa chúng.

Việc khai báo các module kết hợp cũng tương tự như khai báo các module đơn giản. Phần khai báo cũng bao gồm các từ khoá parameters và gates, ngoài ra nó còn sử dụng thêm hai từ khoá là submodules và connections.

Cú pháp:

module Tên_module

parameters:

//...

gates:

//...

submodules:

//...

connections:

//...

endmodule

Chú ý là tất cả các khai báo trên (parameters, gates, submodules, connections) chỉ là tuỳ chọn.

3.5.1. Các tham số và cổng của module kết hợp

Các tham số và cổng của module kết hợp cũng được khai báo và hoạt động tương tự như các tham số và cổng của các module đơn giản.

Các tham số của module kết hợp có thể được sử dụng bởi các module con và thường được dùng để khởi tạo giá trị cho các tham số của các module con.

Các tham số cũng có thể được sử dụng để xác định cấu trúc bên trong của các module kết hợp: số các module con, kích thước của các vector cổng mặt khác các tham số này cũng có thể được sử dụng để xác định các kết nối bên trong module kết hợp.

Các tham số ảnh hưởng đến cấu trúc bên trong của module nên được khai báo là const để giá trị của tham số không thay đổi theo các lần truy nhập. Trái lại nếu các tham số được khai báo là các giá trị ngẫu nhiên, người sử dụng có thể sẽ có các giá trị khác nhau mỗi lần tham số được truy nhập trong quá trình xử lý của module kết hợp.

Ví dụ:

module Router

parameters:

packetsPerSecond : numeric,

bufferSize : numeric,

numOfPorts : const;

gates:

in: inputPort[];

out: outputPort[];

submodules: //...

connections: //...

endmodule

3.5.2. Các module con

Các module con được khai báo sau từ khoá submodules. Theo quy ước các module con được đặt tên bắt đầu với chữ cái thường.

Các module con có thể là một module đơn giản hoặc một module kết hợp. Trình biên dịch NED phải biết được kiểu của module do đó các module con phải được khai báo sớm hơn hoặc được import từ các file NED khác.

Người sử dụng cũng có khả năng tạo ra các vector module con và kích thước của vector này có thể nhận vào từ giá trị của một tham số. Khi khai báo các module con, bạn cần phải gán giá trị cho các tham số của module và nếu kiểu module tương ứng có sử dụng các vector cổng thì bạn phải xác định cho nó một kích thước cụ thể.

Ví dụ:

module Tên_Module_kết_hợp

//...

submodules:

tên_module_con_1: Kiểu_Module_1

parameters:

//...

gatesizes:

//...

tên_module_con_2: Kiểu_Module_2

parameters:

//...

gatesizes:

//...

endmodule

Vector module

Vector module là một tập hợp (một mảng) các module con. Kích thước của vector có thể được biểu diễn bằng một biểu thức đặt trong cặp dấu ngoặc vuông '[]'. Biểu thức này có thể tham chiếu đến các tham số của module. Giá trị 0 cho số các module cũng được chấp nhận.

Ví dụ:

module CompoundModule

parameters:

size: const;

submodules:

submod1: Node[3]

//...

submod2: Node[size]

//...

submod3: Node[2*size+1]

//...

endmodule

3.5.3. Tham số tên kiểu module con

Việc sử dụng tên của các kiểu module như các tham số tạo điều kiện dễ dàng cho sử dụng các module con. Lấy ví dụ, cho rằng mục đích của quá trình mô phỏng là so sánh sự khác nhau giữa các thuật toán tìm đường. Giả sử bạn đã lập trình các thuật toán tìm đường cần thiết như các module đơn giản DistVecRoutingNode, AntNetRouting1Node, AntNetRouting2Node, ... Bạn cũng đã tạo ra một topology mạng như một module kết hợp gọi là RountingTestNetwork để phục vụ cho việc đánh giá hoạt động của các thuật toán. Hiện tại RountingTestNetwork đang sử dụng thuật toán DistVecRoutingNode (tất cả các module con đều có kiểu này) và bạn muốn có

thể chuyển đổi qua lại một cách dễ dàng giữa các thuật toán để tiện lợi cho việc đánh giá.

Để thực hiện điều này ta có thể sử dụng thêm một biến là routingNodeType cho module RoutingTestNetwork. Đồng thời bạn cũng khai báo cho NED các module con của RoutingTestNetwork không có kiểu cố định, mà kiểu của các module này được là giá trị của biến routingNodeType. Khi đó mạng mô phỏng của bạn có thể dễ dàng thay đổi các thuật toán ở trên thông qua giá trị của tham số như "DistVectRoutingNode", "AntNetRouting1Node" hoặc "AntNetRouting2Node". Trong trường hợp giá trị của tham số là sai (chứa tên của một kiểu không tồn tại) quá trình mô phỏng sẽ bị lỗi khi bắt đầu chạy - module type definition not found (khai báo kiểu module không được tìm thấy).

Bên trong module RoutingTestNetwork, ta có thể gán giá trị cho các tham số và tiến hành kết nối với các module chứa các thuật toán tìm đường tương ứng. Tuy nhiên để tăng tính chính xác, đảm bảo tên của tham số và cổng mà bạn sử dụng là chính xác, NED cần có sự giúp đỡ từ bạn. Bạn có thể khai báo thêm một module (giả sử tên là RoutingNode) và phải đảm bảo chắc chắn rằng tất cả các module mà định sử dụng thông qua tham số routingNodeType đều có các tham số và các cổng giống như của module RoutingNode.

Cú pháp:

```
module RoutingTestNetwork
parameters:
routingNodeType: string; // should hold the name
// of an existing module type
gates: //...
submodules:
node1: routingNodeType like RoutingNode;
node2: routingNodeType like RoutingNode;
//...
connections nocheck:
node1.out0 --> node2.in0;
//...
endmodule
```

Đoạn mã này nếu nhìn theo góc độ của ngôn ngữ C++ thì RoutingNode đóng vai trò của một lớp cơ sở, DistVecRoutingNode, AntNetRouting1Node là các lớp kế thừa từ lớp RoutingNode. Tham số routingNodeType tương ứng với con trỏ trỏ tới lớp cơ sở.

Module RoutingNode không cần được thực hiện trong C++ bởi không có đối tượng cụ thể nào của nó được tạo ra, nó chỉ đơn thuần được dùng để kiểm tra tính chính xác của file NED. Mặt khác, các module thực sự sẽ được thay thế (ví dụ như DistVecRoutingNode, AntNetRouting1Node,...) sẽ không cần phải khai báo trong file NED.

Từ khoá like cho phép bạn tạo ra một họ các module phục vụ cho cùng một mục đích, có cùng giao tiếp giống nhau (có cùng các tham số và các cổng) và sử dụng chúng thay thế nhau trong file NED.

3.5.4. Gán giá trị cho các tham số của các module con

Có thể gán giá trị cho các tham số của các module con trong phần khai báo parameters của các module con. Các tham số của module con có thể được gán giá trị như các hằng số hoặc có thể sử dụng ngay các tham số của module kết hợp chứa nó, hoặc cũng có khởi gán bằng một biểu thức.

Không bắt buộc tất cả các tham số đều phải khởi gán giá trị. Giá trị của tham số có thể nhận trong lúc thực hiện hoặc nhận từ file cấu hình hoặc trong trường hợp giá trị của tham số không có trong file cấu hình, quá trình mô phỏng sẽ nhắc bạn. Tuy nhiên nếu các tham số để trong file cấu hình, sẽ dễ dàng hơn cho việc sửa chữa giá trị của các tham số.

Ví dụ:

module CompoundModule

parameters:

param1: numeric,

param2: numeric,

useParam1: bool;

submodules:

submodule1: Node

parameters:

p1 = 10,

p2 = param1+param2,

p3 = useParam1==true ? param1 : param2;

//...

endmodule

Trong khi mô hình hoạt động, các biểu thức gán giá trị vẫn được tính toán nếu các tham số tương ứng được gọi đến. Ngoài ra để gọi một tham số của module con ta có thể sử dụng cú pháp như sau: submodule.parametername (hoặc submodule[index].parametername).

Từ khoá input

Khi một tham số không nhận giá trị trực tiếp trong file NED hoặc trong file cấu hình, người sử dụng sẽ được nhắc để nhập giá trị cho tham số khi quá trình mô phỏng bắt đầu thực hiện. Tuy nhiên nếu bạn muốn chủ động nhập giá trị tham số khi bắt đầu quá trình mô phỏng, bạn có thể sử dụng từ khoá input. Từ khoá input cũng cho phép người sử dụng có thể thiết lập thông báo nhập giá trị hay đặt giá trị mặc định cho tham số.

Cú pháp:

parameters:

numCPUs = input(10, "Number of processors?"), //giá trị mặc

//định, dấu nhắc

processingTime = input(10ms), //thông báo nhập giá trị

cacheSize = input;

3.5.5. Khai báo kích thước của các vector cổng của module con

Kích thước của các vector cổng được khai báo bằng từ khoá gatesizes. Kích thước này có thể được khai báo như một hằng số, một tham số hay một biểu thức.

Ví dụ:

simple Node

gates:

in: inputs[];

out: outputs[];

endsimple

module CompoundModule

parameters:

numPorts: const;

submodules:

node1: Node

gatesizes:

inputs[2], outputs[2];

node2: Node

gatesizes:

inputs[numPorts], outputs[numPorts];

//...

endmodule

gatesizes là không bắt buộc, nếu bạn muốn bỏ qua việc khai báo gatesizes cho vector cổng nó sẽ được đặt bằng 0. Một lý do để bỏ qua việc gán giá trị cho gatesizes là bạn sẽ sử dụng gate++ ("extend gate vector with a new gate" - vector cổng mở rộng với một cổng mới). gate++ sẽ được trình bầy kĩ hơn trong phần Connection.

3.5.6. Khai báo gatesizes và tham số có điều kiện

Kích thước của vector cổng và các tham số trong module con thường được khai báo kèm thêm điều kiện.

Ví dụ:

module Chain

parameters: count: const;

submodules:

node : Node [count]

parameters:

position = "middle";

parameters if index==0:

position = "beginning";

parameters if index==count-1:

position = "end";

gatesizes:

in[2], out[2];

gatesizes if index==0 || index==count-1:

in[1], out[1];

connections:

//...

endmodule

Chú ý các giá trị mặc định nên được khai báo đầu tiên bởi vì NED sẽ duyệt từ trên xuống dưới, nếu gặp điều kiện đúng thì các giá trị tương ứng sẽ được chèn vào các giá trị mặc định trước đó. Trong trường hợp khai báo giá trị mặc định cuối cùng, giá trị mặc định sẽ có thể chèn vào giá trị của một trường hợp điều kiện đúng trước đó.

3.5.7. Kết nối

Các kết nối chỉ ra cụ thể cách các cổng của module kết hợp giao tiếp với các cổng tương ứng của module con.

Kết nối có thể được tạo ra giữa hai module con hoặc giữa module con với module cha (module kết hợp) trực tiếp chứa nó (trong một số ít trường hợp, một kết nối cũng có thể được tạo ra giữa hai cổng của cùng một module kết hợp). Điều này có nghĩa là NED không cho phép một kết nối đa cấp (kết nối giữa hai module xa nhau trong cấu trúc phân cấp). Hạn chế này làm tăng tính độc lập và khả năng dùng lại của mỗi module. Ngoài ra, hướng của module cũng rất trong quan trọng khi tạo kết nối. Không thể tạo một kết nối giữa hai cổng ra hoặc giữa hai cổng vào với nhau.

NED chỉ hỗ trợ kiểu kết nối một-một do đó một cổng riêng biệt được sử dụng chỉ xuất hiện một lần trong một kết nối. Kiểu kết nối một-nhiều và nhiều-một cũng có thể được tạo ra bằng cách sử dụng các module đơn giản trong đó các luồng message được nhân đôi hoặc được ghép thêm (duplicate message or merge message flows).

Các kết nối được liệt kê sau từ khoá connections và được phân tách với nhau bằng dấu chấm phẩy.

Ví dụ:

module CompoundModule

parameters: //...

gates: //...

submodules: //...

connections:

node1.output --> node2.input;

node1.input <-- node2.output;

//...

endmodule

Cổng nguồn có thể là cổng ra của các module con hoặc là cổng vào của module kết hợp và cổng đích có thể là cổng vào của module con hay cổng ra của module kết hợp. Mũi tên có thể chỉ theo chiều từ trái qua phải hoặc theo chiều ngược lại.

Chú thích gate++ cho phép người sử dụng có thể một vector cổng với một cổng mới, mà không cần phải khai báo trong gatesizes.

Ví dụ:

simple Node

gates:

in: in[];

out: out[];

endsimple

module SmallNet

submodules:

node: Node[6];

connections:

node[0].out++ --> node[1].in++;

node[0].in++ <-- node[1].out++;

node[1].out++ --> node[2].in++;

node[1].in++ <-- node[2].out++;

node[1].out++ --> node[4].in++;

node[1].in++ <-- node[4].out++;

node[3].out++ --> node[4].in++;

node[3].in++ <-- node[4].out++;

node[4].out++ --> node[5].in++;

node[4].in++ <-- node[5].out++;

endmodule

Một kết nối:

- Có thể có các thuộc tính (độ trễ, tỉ số bit lỗi, tốc độ truyền dữ liệu) hoặc sử dụng một kênh truyền đã được đặt tên.

- Có thể xuất hiện trong một vòng lặp (để tạo ra nhiều kết nối).
- Có thể là điều kiện.

Kết nối đơn và kênh truyền

Nếu bạn không xác định một kênh truyền, thì kết nối sẽ không có trễ và không có bit lỗi khi truyền. Bạn có thể xác định một kênh truyền thông qua tên.

Ví dụ:

node1.outGate --> Fiber --> node2.inGate;

Trong trường hợp này file NED đã phải có khai báo loại kênh truyền trên. Hoặc người sử dụng cũng có thể xác định trực tiếp một kênh truyền qua các tham số đặc trưng.

Ví dụ:

node1.outGate --> error 1e-9 delay 0.001 --> node2.inGate;

Không nhất định phải khai báo đầy đủ các tham số và các tham số có thể được khai báo theo bất kỳ thứ tự nào.

Kết nối vòng lặp

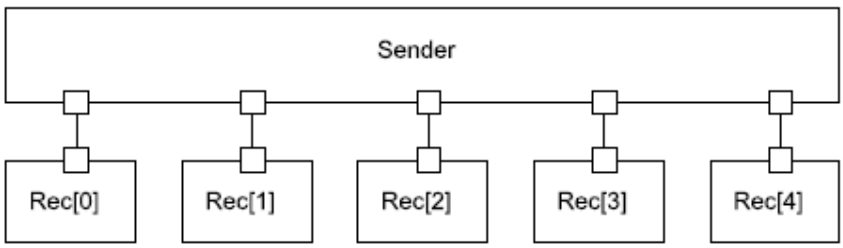
Nếu module con hoặc các vector cổng được sử dụng, NED cho phép người sử dụng có thể tạo ra nhiều hơn một kết nối với một khai báo. Đó được gọi là một đa kết nối hay một kết nối vòng lặp.

for i=0..4 do

node1.outGate[i] --> node2[i].inGate

endfor;

Một đa kết nối thực chất là một tập hợp các kết nối đơn được khai báo gọn hơn nhờ câu lệnh vòng lặp for...do...endfor.



Hình I-3.1 - Đa kết nối

Các kết nối có điều kiện

Người sử dụng có thể sử dụng từ khoá if để ràng buộc điều kiện khi tạo các kết nối.

Ví dụ:

for i=0..n do

node1.outGate[i] --> node2[i].inGate if i%2==0;

endfor;

Nocheck

Mặc định, NED quy định tất cả các cổng đều phải được kết nối do vậy trong nhiều trường hợp việc kiểm tra này có thể gây nhiều phiền phức. Để tắt bỏ chức năng này bạn có thể sử dụng từ khoá nocheck.

3.6. Khai báo mạng

Để thực sự tạo một mô hình mô phỏng chạy được thì người sử dụng phải khai báo mạng. Việc khai báo mạng sẽ tạo ra một mô hình mô phỏng như là một đối tượng cụ thể của một kiểu module đã định nghĩa trước đó. Kiểu module ở đây thường là một module kết hợp, tuy nhiên cũng có thể tạo ra một mạng chỉ là một module đơn giản độc lập.

Có thể khai báo nhiều mạng trong một hoặc nhiều file NED. Chương trình mô phỏng sử dụng các file NED đó sẽ có thể chạy bất cứ một mạng nào. Nếu bạn muốn cụ thể một mạng nào đó được thực hiện bạn có thể chỉ rõ trong file cấu hình (omnetpp.ini).

Cú pháp khai báo mạng cũng tương tự như khai báo các module con:

network wirelessLAN: WirelessLAN

parameters:

numUsers=10,

httpTraffic=true,

ftpTraffic=true,

distanceFromHub=truncnormal(100,60);

endnetwork

Ở đây WirelessLAN là tên của một kiểu module kết hợp đã định nghĩa từ trước, trong đó có thể chứa các kiểu module kết hợp khác như WirelessHost, WirelessHub... Một cách tự nhiên, chỉ các kiểu module không có cổng mới có thể được dùng trong các khai báo mạng.

3.7. Các biểu thức

Các biểu thức được sử dụng trong NED được viết theo cú pháp của C++. Các biểu thức dùng các toán tử của C++, có thể sử dụng các tham số theo cả hai hình thức truyền theo tham trị hoặc tham biến, có thể gọi các hàm của C++, nhận các giá trị ngẫu nhiên hoặc yêu cầu nhập từ người sử dụng...

Khi một tham số được gán trị bằng một biểu thức thì giá trị biểu thức đó chỉ được tính mỗi lần tham số được gọi tới (trừ khi tham số được khai báo const). Điều này có nghĩa là một module đơn giản sử dụng một tham số không được khai báo const sẽ nhận được các giá trị khác nhau mỗi lần gọi tham số trong quá trình mô phỏng. Các biểu thức khác (chứa các tham số được khai báo const) sẽ chỉ được tính một lần. Các tham số có kiểu XML có thể được dùng để truy nhập vào các file XML hay một phần nào đó của các file này. Tham số kiểu XML có thể được gán giá trị bằng toán tử xmldoc().

3.7.1. Hằng số

Hằng số học

Hằng số học thường là các số dạng thập phân hoặc các số thập phân với dấu phẩy động.

Hằng chuỗi ký tự

Hằng chuỗi ký tự được khai báo giữa cặp dấu ngoặc kép.

Hằng thời gian

Để sử dụng hằng thời gian bạn phải sử dụng thêm các từ khoá chỉ đơn vị thời gian.

Ví dụ:

...

parameters:

propagationDelay = 560ms, // 0.560 giây

connectionTimeout = 6m 30s 500ms, // 390.5 giây

recoveryIntvl = 0.5h; // 30 phút

Các đơn vị thời gian có thể được sử dụng:

Đơn vị	Ý nghĩa
ns	nano giây
us	micro giây
ms	mili giây
s	giây
m	phút (60 giây)
h	giờ (3600 giây)
d	ngày (86400 giây)

3.7.2. Tham chiếu

Các biểu thức có thể sử dụng các tham số của module kết hợp trực tiếp chứa nó và của các module đơn giản đã được khai báo trước đó trong file NED.

Cú pháp: submod.param hoặc submod[index].param.

Có hai từ khoá có thể được sử dụng với tên của tham số: ancestor và ref.

ancestor có nghĩa là nếu một module kết hợp không chứa một tham số nào, thì tham số đó sẽ được tìm kiếm trong các module cấp cao hơn trong cấu trúc phân cấp. ancestor không được khuyến khích sử dụng bởi vì nó xâm phạm tới khái niệm đóng

gói thông tin (encapsulation) và có thể chỉ được kiểm tra trong thời gian chạy. Nó chỉ nên được sử dụng trong một số ít những trường hợp thực sự cần thiết.

ref lấy giá trị của tham số bằng phương pháp tham chiếu, có nghĩa là việc thay đổi giá trị của tham số trong thời gian chạy sẽ gây ảnh hưởng tới tất cả các module tham chiếu tới tham số này. Cũng giống như ancestor, ref nên được sử dụng hạn chế. Một trường hợp có thể sử dụng ref là khi phải điều chỉnh mô hình trong thời gian chạy để tìm điều kiện tối ưu. Người sử dụng có thể khai báo một tham số ở mức cao nhất của mô hình và đặt các module khác tham chiếu tới tham số này. Khi bạn thay đổi tham số này trong thời gian chạy, nó sẽ ảnh hưởng tới toàn bộ mô hình. Trong một số trường hợp khác, các tham số được tham chiếu có thể được dùng như các biến trạng thái đối với các module bên cạnh.

3.7.3. Các toán tử

Các toán tử được hỗ trợ trong NED cũng tương tự như các toán tử trong C/C++, tuy nhiên cũng có một số khác biệt:

Dấu ^ được dùng cho phép tính luỹ thừa (không phải là phép XOR các bit như trong C).

Dấu # được sử dụng cho phép toán logic XOR (tương tự như dấu !=) và ## được dùng cho phép toán bit XOR.

Thứ tự ưu tiên của các phép toán bit (&, |, #) là cao nhất so với các toán tử quan hệ khác.

Tất cả các biến trong NED đều có kiểu doubles. Đối với các toán tử bit, kiểu doubles được chuyển thành kiểu unsigned long bằng hàm chuyển đổi có sẵn của C/C++ (type cast), sau khi phép toán được thực hiện kết quả sẽ được chuyển đổi lại thành kiểu doubles. Tương tự đối với các toán tử logic &&, || và ##, các toán hạng sẽ được chuyển sang kiểu bool (type cast) và sau đó kết quả sẽ lại được đổi về kiểu doubles. Đối với phép chia lấy phần dư (%), toán hạng sẽ được chuyển sang kiểu long.

Danh sách các toán tử và thứ tự ưu tiên:

Toán tử	Ý nghĩa
-, !, ~	dấu âm, phủ định, lấy phần bù của bit
^	phép toán luỹ thừa
*, /, %	phép nhân, chia, chia lấy phần dư
+, -	phép cộng, trừ
<<, >>	phép dịch bit
&, |, #	phép toán bit and, or, xor
==	so sánh bằng
!=	so sánh khác

>, >=	so sánh lớn hơn, lớn hơn hoặc bằng
<, <=	so sánh nhỏ hơn, nhỏ hơn hoặc bằng
&&, ||, ##	toán tử logic and, or, xor
?:	toán tử "inline if"

3.7.4. Toán tử sizeof() và index

Toán tử sizeof() trả về kích thước của một vector cổng. Toán tử index trả về chỉ số của module con hiện thời trong một vector module.

Ví dụ dưới đây mô tả một router với một số cổng và một đơn vị routing (giả sử các vector cổng in[] và out[] có cùng kích thước).

```
module Router
gates:
in: in[];
out: out[];
submodules:
port: PPPInterface[sizeof(in)]; // one PPP for each input
                        // gate
parameters: interfaceId = 1+index; // 1,2,3...
routing: RoutingUnit;
gatesizes:
in[sizeof(in)]; // one gate pair for each port
out[sizeof(in)];
connections:
for i = 0..sizeof(in)-1 do
in[i] --> port[i].in;
out[i] <-- port[i].out;
port[i].out --> routing.in[i];
port[i].in <-- routing.out[i];
endfor;
endmodule
```

3.7.5. Toán tử xmldoc()

Toán tử xmldoc() có thể được dùng để gán giá trị cho các tham số kiểu XML, nghĩa là có thể dùng toán tử này để trỏ vào các file XML hay các element cụ thể trong file XML. Có thể sử dụng toán tử xmldoc() theo hai cách:

xmldoc() nhận vào tên file.
Ví dụ: xmlparam = xmldoc(“someconfig.xml”);

xmldoc() nhận vào tên file cộng thêm XPath, biểu thức chỉ ra một element cụ thể được chọn trong file XML.
Ví dụ: xmlparam = xmldoc("someconfig.xml", "/config/profile[@id=’2’]");

3.7.6. XML và XPath

Tham số XPath của toán tử xmldoc() chỉ ra một element cụ thể trong file XML, điều này cho phép dễ dàng nối nhiều file cấu hình dạng XML nhỏ thành một file lớn hơn. Nếu biểu thức XPath phù hợp với nhiều element trong file thì element đầu tiên (thứ tự duyệt theo chiều sâu) sẽ được chọn.

Cú pháp của biểu thức:

Biểu thức chứa đường dẫn trong đó các thành phần được phân tách bằng dấu “/” hoặc “//”.

Các thành phần của đường dẫn có thể là tên thẻ (tag name) của các element, dấu “*”, “.” hoặc “..”. Tên thẻ của các element và dấu “*” có thể có thêm biểu thức mô tả thuộc tính của element theo dạng “[vị trí]” hoặc “[@thuộc_tính=’giá trị’]”. Vị trí của các element trong file XML được tính bắt đầu từ 0.

Dấu “/” có nghĩa là xét các element con; dấu “//” sẽ xét đến các element ở bất kỳ cấp nào nằm dưới element hiện thời.

Dấu “.”, “..” và “*” lần lượt đại diện cho element hiện thời, element cha và một element với tên bất kỳ.

Ví dụ:

/foo – element gốc (root element) có tên là <foo>

/foo/bar – element con <bar> đầu tiên của element gốc <foo>

//bar – element <bar> đầu tiên ở bất kỳ cấp nào (duyệt theo chiều sâu)

/*/bar – element con <bar> đầu tiên của element gốc có tên bất kỳ

/*/*/bar – element con <bar> đầu tiên dưới hai cấp của element gốc

/*/foo[0] – element con <foo> đầu tiên của element gốc

/*/foo[1] – element con <foo> thứ hai của element gốc

/*/foo[@color=’green’] – element con <foo> đầu tiên có thuộc tính “color” có giá trị bằng “green”

//bar[1] – một element <bar> ở vị trí bất kỳ nhưng phải là element <bar> thứ hai

//*[@color=’yellow’] – bất kỳ element nào ở bất kỳ vị trí nào có thuộc tính "color" có giá trị bằng "yellow"

//*[@color=’yellow’]/foo/bar – element con <bar> đầu tiên của element con <foo> đầu tiên của một element có thuộc tính “color” bằng “yellow” ở vị trí bất kỳ.

3.7.7. Hàm

Trong NED, bạn có thể sử dụng các hàm toán học sau:

Rất nhiều hàm có trong thư viện toán của C (math.h) như exp(), log(), sin(), cos(), floor(), ceil()...

Các hàm tạo giá trị ngẫu nhiên: uniform, exponential, normal...

3.7.8. Giá trị ngẫu nhiên

Các tham số trừ khi được khai báo là const, nếu không giá trị của các tham số thường là các giá trị ngẫu nhiên. Các giá trị này là khác nhau mỗi lần tham số được gọi đến. Các giá trị ngẫu nhiên được sinh ra nhờ bộ tạo số ngẫu nhiên (Random Number Generator - RNG) của OMNeT++.

Hàm	Mô tả
Luật phân phối liên tục	
uniform(a, b, rng=0)	luật phân phối đều trong khoảng [a, b)
exponential(mean, rng=0)	luật phân phối theo luật số mũ với giá trị trung bình mean
normal(mean, stddev, rng=0)	luật phân phối bình thường với giá trị trung bình mean và độ lệch chuẩn stddev
truncnormal(mean, stddev, rng=0)	luật phân phối bình thường loại bỏ các số không âm
gamma_d(alpha, beta, rng=0)	luật phân phối gamma với alpha > 0 và beta > 0
beta(alpha1, alpha2, rng=0)	luật phân phối beta với alpha1>0 và alpha2>0
erlang_k(k, mean, rng=0)	luật phân phối Erlang với k>0 pha và giá trị trung bình mean
chi_square(k, rng=0)	luật phân phối chi_square với k>0 độ tự do
student_t(i, rng=0)	luật phân phối student_t với i>0 độ tự do
cauchy(a, b, rng=0)	luật phân phối Cauchy với các tham số a, b trong b>0
triang(a, b, c, rng=0)	luật phân phối tam giác với các tham số a<=b<=c, a!=c
lognormal(m, s, rng=0)	luật phân phối lognormal với giá trị trung bình m và độ sai khác s > 0
weibull(a, b, rng=0)	luật phân phối Weibull với các tham số a>0, b>0
pareto_shifted(a, b, c, rng=0)	luật phân phối Pareto tổng quát với các tham số a,

	b và độ dịch c.
Luật phân phối rời rạc	
intuniform(a, b, rng=0)	luật phân phối đều với các số nguyên nằm trong khoảng a..b
bernoulli(p, rng=0)	kết quả của phép thử Bernoulli với xác suất p, 0<=p<=1 (bằng 1 với xác suất p và bằng 0 với xác suất (1-p))
binomial(n, p, rng=0)	luật phân phối binomial với tham số n>=0 và 0<=p<=1
geometric(p, rng=0)	luật phân phối geometric với tham số 0<=p<=1
negbinomial(n, p, rng=0)	luật phân phối binomial với tham số n>0 và 0<=p<=1
poisson(lambda, rng=0)	luật phân phối Poisson với tham số lambda

3.7.9. Khai báo một hàm mới

Các hàm do người dùng định nghĩa được lập trình dưới dạng các hàm trong C++. Các hàm này có thể có từ 0 đến 5 tham số, tuy nhiên cả tham số và kết quả trả về đều phải có kiểu double. Các hàm này phải được đăng ký trong một trong các file của C++ với macro Define_Function().

Ví dụ (chú ý đoạn code dưới đây phải nằm trong một file C++ xác định):

```
#include <omnetpp.h>

double average(double a, double b)

{

return (a+b)/2;

}

Define_Function(average, 2);
```

Số 2 chỉ ra rằng hàm average nhận hai tham số. Sau khi đăng ký, hàm average có thể được sử dụng trong các file NED.

```
module Compound

parameter: a,b;

submodules:

proc: Processor

parameters: av = average(a,b);

endmodule
```

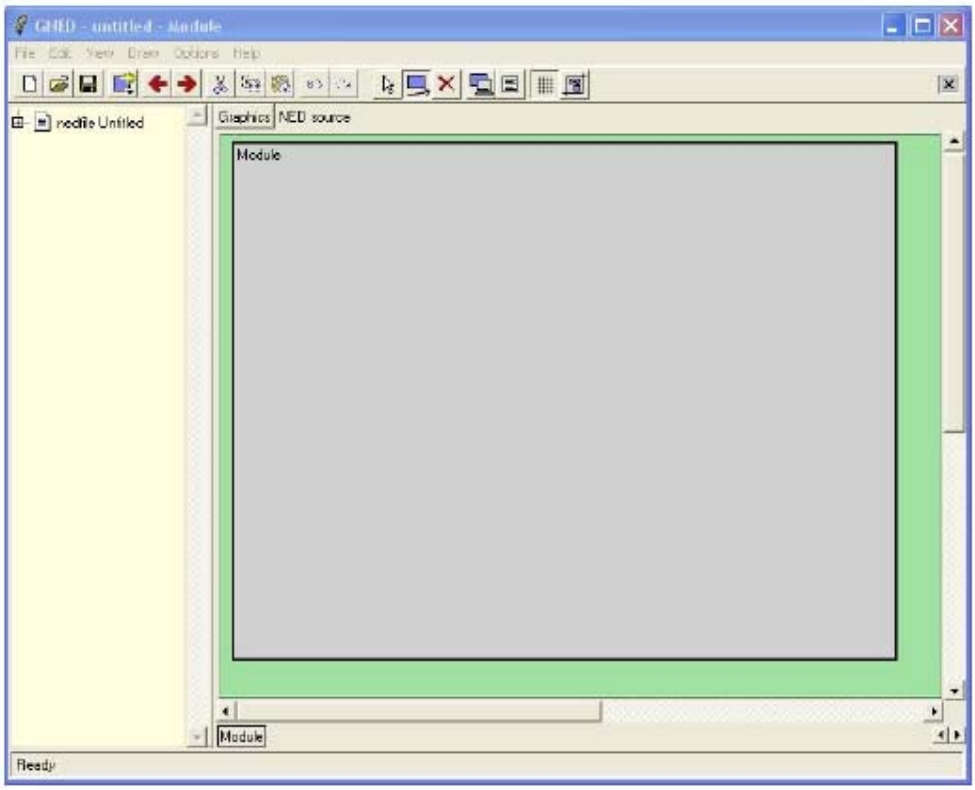
Nếu hàm nhận vào tham số có kiểu int, long hoặc một số kiểu khác không phải là kiểu double thì bạn phải tạo hàm chuyển đổi. Trong trường hợp này bạn phải đăng ký hàm chuyển đổi với macro Define_Function2() để cho phép một hàm có thể được đăng ký

một tên khác với tên mà hàm được gọi khi sử dụng. Trong trường hợp hàm không trả về kiểu double bạn cũng có thể làm tương tự.

```
#include <omnetpp.h>

long factorial(int k)

{

...

}

static double _wrap_factorial(double k)

{

return factorial((int)k);

}

Define_Function2(factorial, _wrap_factorial, 1);
```

4. GIỚI THIỆU GNED

4.1. Giao diện



Giao diện của GNED hỗ trợ hai chế độ:

- Đồ hoạ (Graphics): là giao diện mặc định.
- Mã nguồn (NED Source): cho phép người sử dụng có thể sửa lại mã nguồn một cách trực tiếp.

Hai kiểu giao diện này có thanh Command Bar là khác nhau. Đối với chế độ đồ hoạ, thanh Command Bar có dạng:

 New NED file: tạo một file NED mới. GNED cho phép tạo nhiều file NED cùng một lúc.

 Open NED file: mở một file NED.

 Save Document: Ghi file NED hiện thời.

 Add new component to current NED file: một danh sách thả xuống (dropdown list) cho phép người sử dụng một trong 5 thành phần có thể thêm vào file NED: Import, Channel, Simple Module, Compound Module và Network.

 Back to previous view: quay lại chế độ giao diện trước trong History.

 Forward to next view: chuyển tới chế độ giao diện tiếp theo trong History.

 Select, move or resize items button: khi nút này được kích hoạt, nó cho phép người sử dụng có thể chọn, di chuyển và thay đổi kích thước các đối tượng trong chế độ đồ hoạ.

 Draw submodules and connection tool: Công cụ tạo các module con và các kết nối. Kích chuột và kéo rê để tạo các module con. Kích chuột bên trong một module con kéo rê sang module khác để tạo kết nối một chiều giữa hai module. Để tạo một kết nối hai chiều ta thực hiện việc tạo kết nối một chiều hai lần theo chiều ngược nhau.

 Appearance of selected items: Chức năng hoạt động khi một (và chỉ một) đối tượng được chọn. Nó sẽ làm xuất hiện các trình đơn như Submodule Appearance, Module Appearance hay Connection Appearance. Các trình đơn ngữ cảnh này cho phép bạn mô tả chi tiết cho đối tượng được chọn.

 Properties of selected items: thuộc tính của đối tượng được chọn.

 Snap to grid on/off switch: bật/tắt chế độ bắt dính vào lưới điểm.

 Fit compound module to area content: ấn nút này sẽ làm cho biên của module chứa được mở rộng ra chứa trọn vẹn biên của tất cả các module con.

Trong chế độ NED Source, thanh Command Bar có dạng:

Bốn nút đầu có chức năng tương tự như trong chế độ Graphic.

 Back to previous module: kích hoạt module đã chọn trước đó.

 Forward to next module: kích hoạt module tiếp theo.

 Cut: di chuyển vào clipboard.

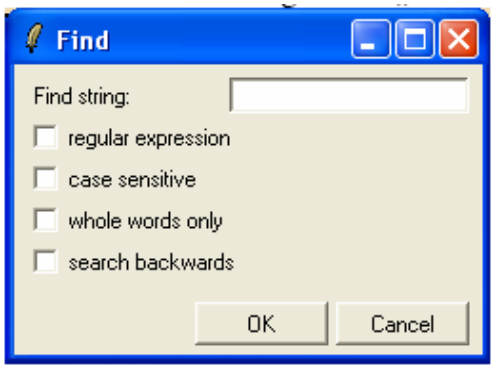
 Copy: sao chép vào clipboard.

 Paste: “dán” từ clipboard.

 Undo: huỷ bỏ thao tác vừa thực hiện.

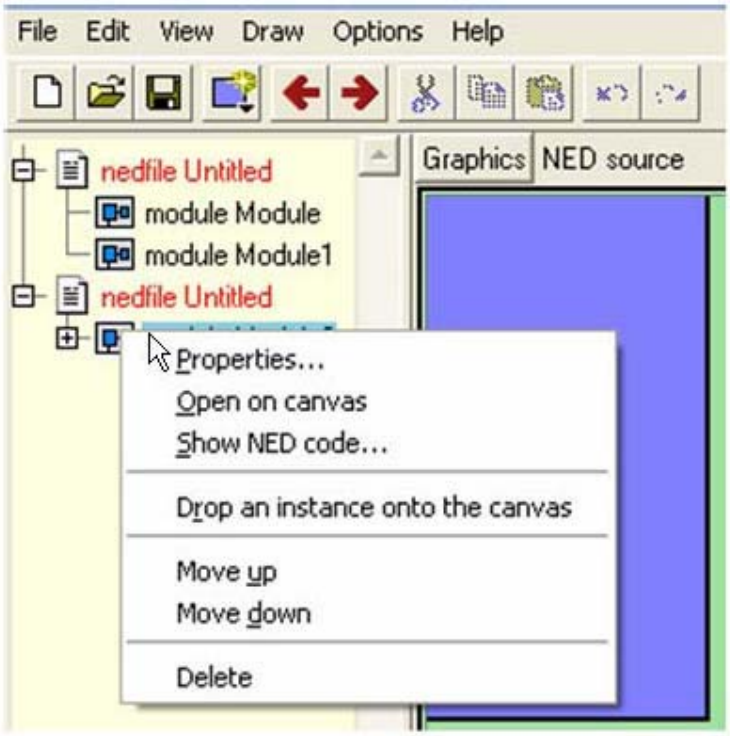
 Redo: lặp lại thao tác vừa huỷ bỏ.

 Find Text: tìm kiếm trong văn bản. Chức năng này sẽ làm xuất hiện hộp thoại:



Hộp thoại cho phép tìm kiếm với các chức năng (bộ lọc): regular expression, case sensitive...

Phần bên trái của giao diện là Tree View. Tree View cho phép quan sát tất cả các file và các module đang được mở.



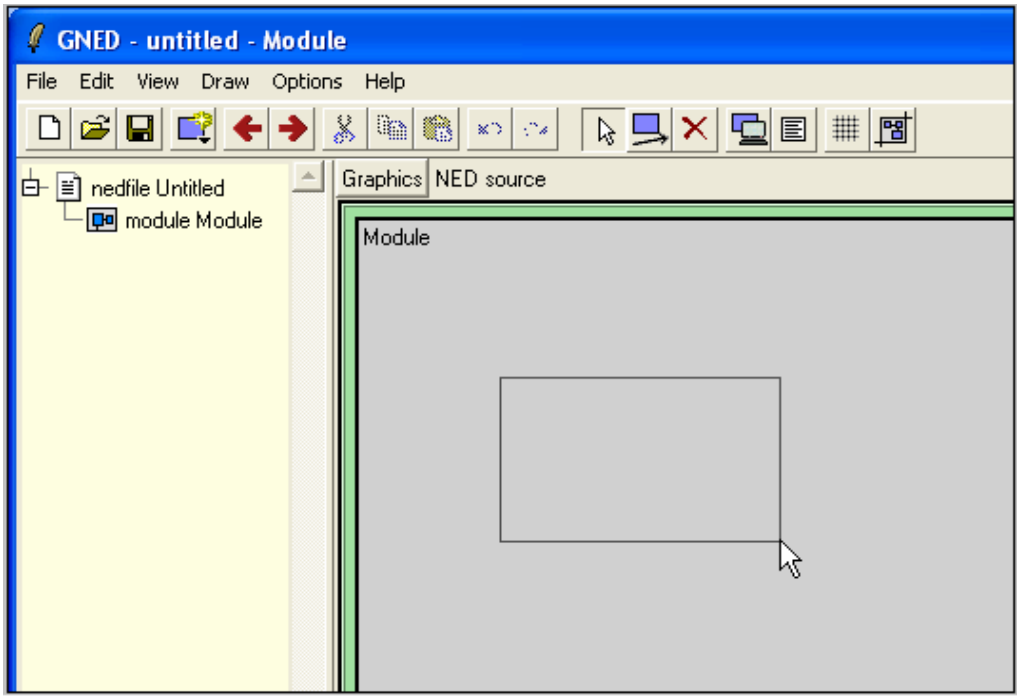
Hai mục "nedfile Untitled" được đánh dấu mầu đỏ để nhắc nhở là hai file này chưa Save.

Nếu bạn kích phải chuột vào một mục nào đó trong Tree View, một trình đơn ngữ cảnh sẽ hiện ra. Trình đơn này chứa các thao tác cơ bản mà bạn có thể thao tác với đối tượng tương ứng.

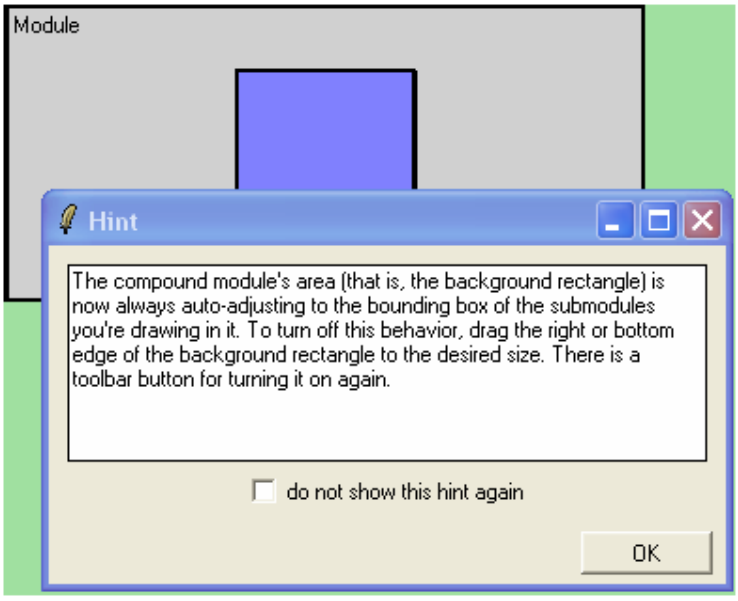
Khi bạn mới sử dụng GNED, sẽ thường có một cửa sổ nhỏ hiện lên mỗi lần bạn thực hiện một thao tác. Cửa sổ này giải thích cho bạn cách thức hoạt động của một số chức năng phức tạp trong giao diện của GNED. Bạn có tắt chức năng này bằng cách đánh dấu vào hộp chọn phía dưới của cửa sổ. Đây là một chức năng rất hữu ích vì vậy khi bạn đã tắt chức năng này đi, muốn nó xuất hiện trở lại bạn phải xoá file .gnedrc.

4.2. Một số thao tác cơ bản

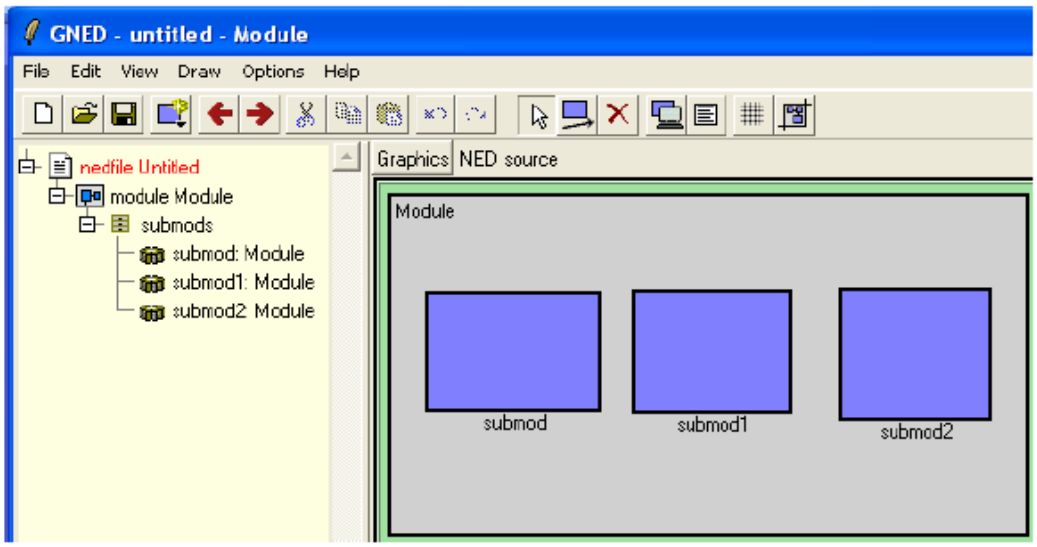
Tạo một module con trong module kết hợp



Kích thước của module kết hợp sẽ tự động điều chỉnh cho phù hợp với module con được tạo ra. Như đã nói ở trên nếu bạn là một người sử dụng mới, sẽ có một cửa sổ hướng dẫn nhỏ xuất hiện:

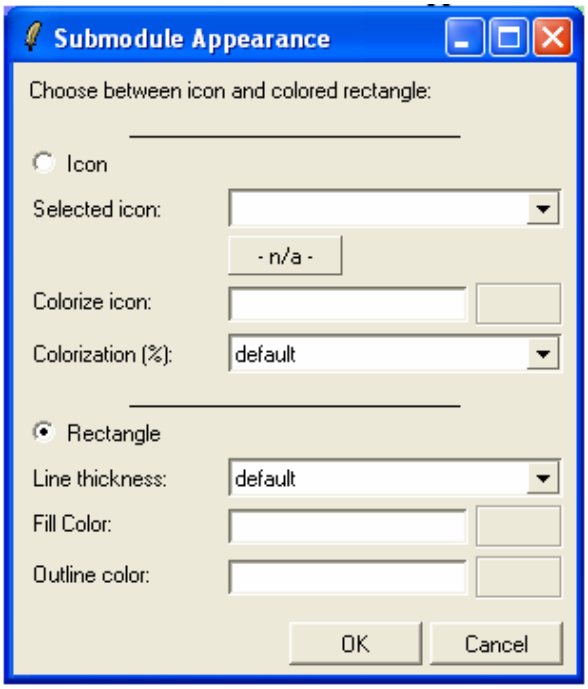


Nhấn OK, tương tự bạn tạo thêm một vài module như trong hình vẽ:

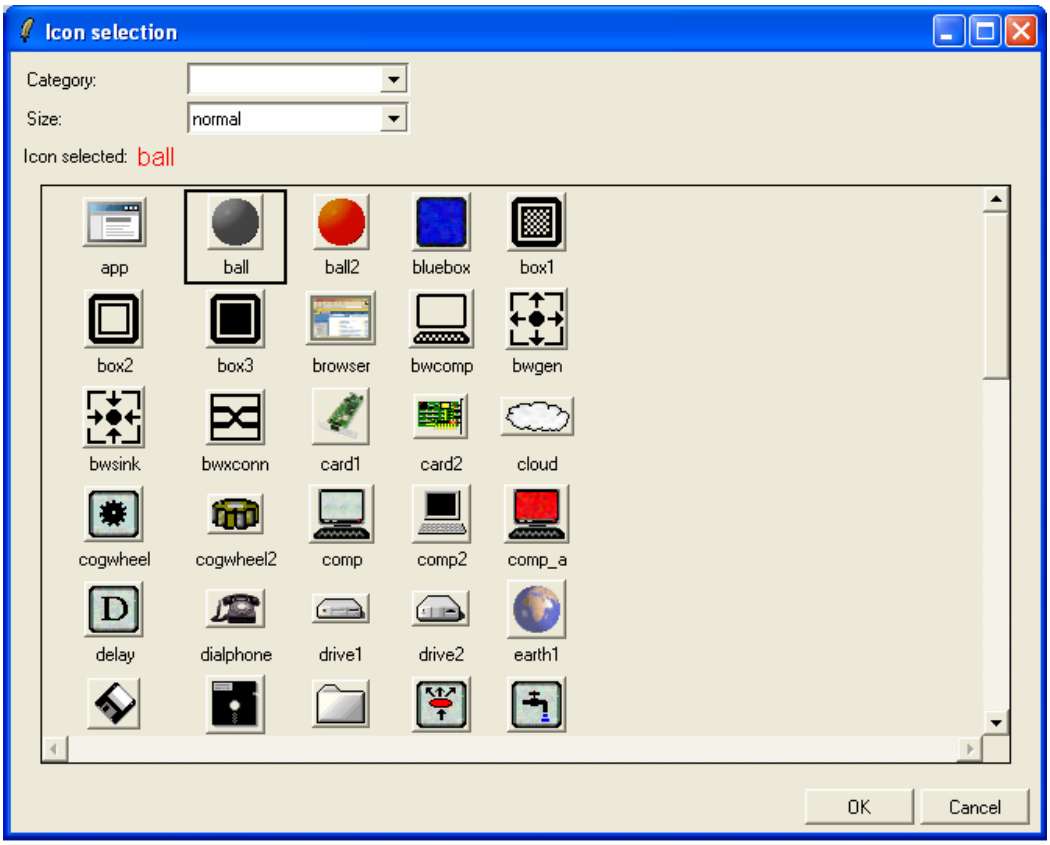


Tiếp theo ta sử dụng công cụ  để chọn một trong các module con vừa được tạo ra. Biên của module được chọn chuyển thành mầu đỏ.

Ấn nút  , bạn sẽ thấy trình đơn ngữ cảnh Submodule Appearance:



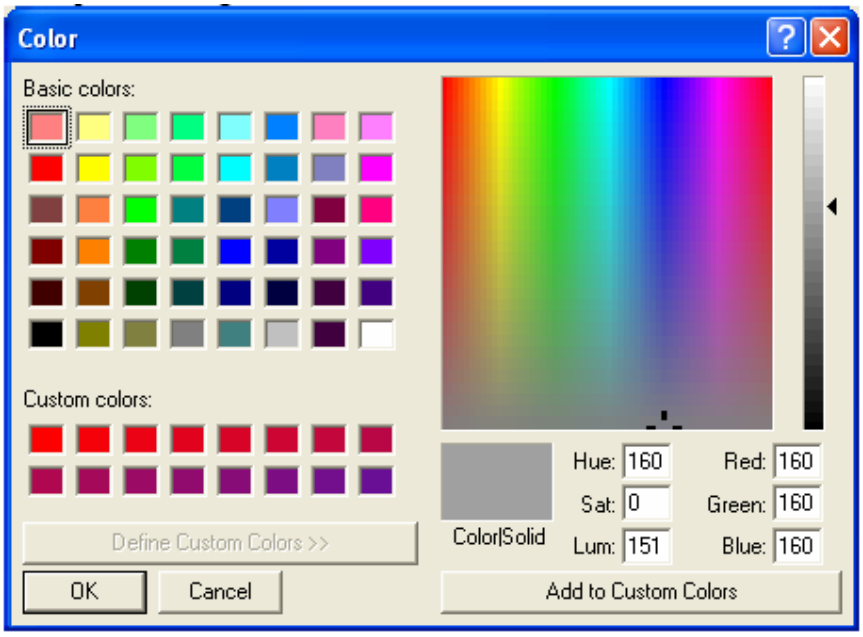
Trình đơn này cho phép người sử dụng có thể thay đổi hình dạng của module con tương ứng. Kích chọn chức năng Icon để chọn một Icon biểu diễn cho module qua trình duyệt đồ hoạ.



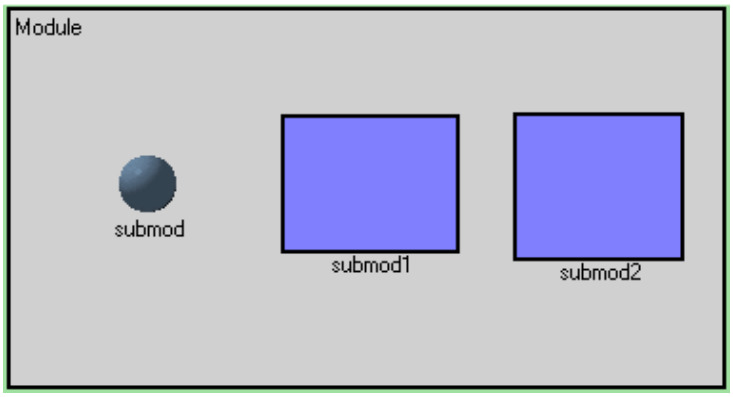
Bạn có thể sử dụng những Icon có sẵn của OMNeT++ hoặc có thể thêm các Icon của riêng mình bằng cách copy các file bitmap vào thư mục bitmap của OMNeT++.

Chọn Icon "ball" với kích thước trung bình ("normal"). Ta cũng có thể thay đổi mầu sắc cho các Icon.

Chuyển Icon "ball" từ mầu xám sang mầu xanh: kích vào hình chữ nhật bên cạnh ô Colorize icon, một bảng chọn mầu sẽ hiện ra

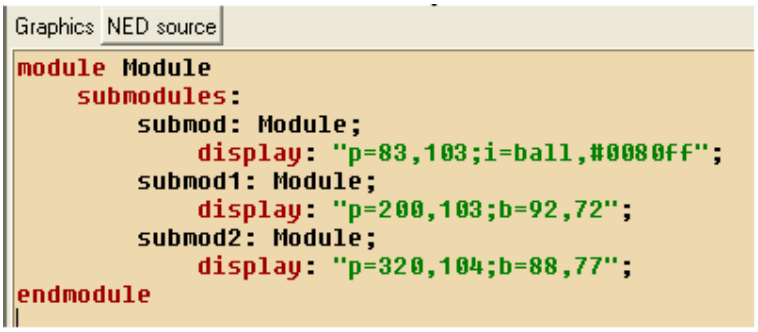


Sau khi chọn mầu xanh, bây giờ bạn có thể xem lại hình dạng của module con mà bạn vừa mới thay đổi so với hình chữ nhật của các module cũ.



Chuyển sang chế độ nhìn NED Source

Bạn có thể thấy mã nguồn của mô hình mà bạn vừa tạo:



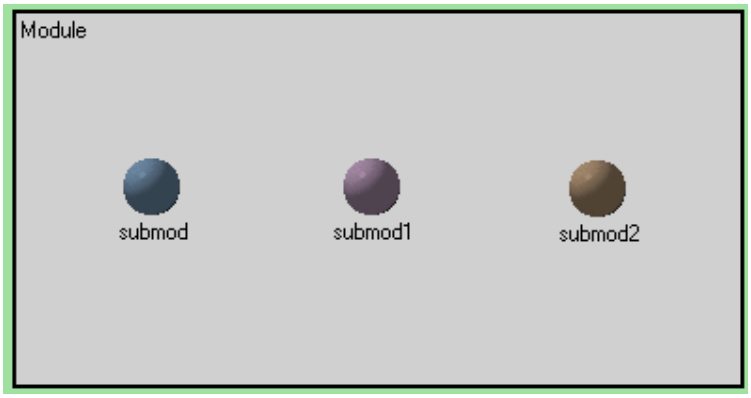
Chú ý là đối với module con thứ nhất những thay đổi về mặt hình thức của nó được diễn tả trong chuỗi "i=ball, #0080ff". Bằng cách sử dụng các chức năng copy, paste ta

có thể thay đổi hình thức cho các module con còn lại mà không cần phải mở trình đơn Submodule Appearance cho từng module tương ứng.

Graphics NED source

```
module Module
    submodules:
        submod: Module;
            display: "p=83,103;i=ball,#0080ff";
        submod1: Module;
            display: "p=200,103;b=92,72;i=ball,#0080ff";
        submod2: Module;
            display: "p=320,104;b=88,77;i=ball,#0080ff";
endmodule
```

Để sinh động hơn, ta thay đổi tham số mầu sắc cho các module còn lại, "submod1" thành "#ff80ff" và "submod2" thành "#ff8000". Kết quả được thể hiện ở hình dưới:

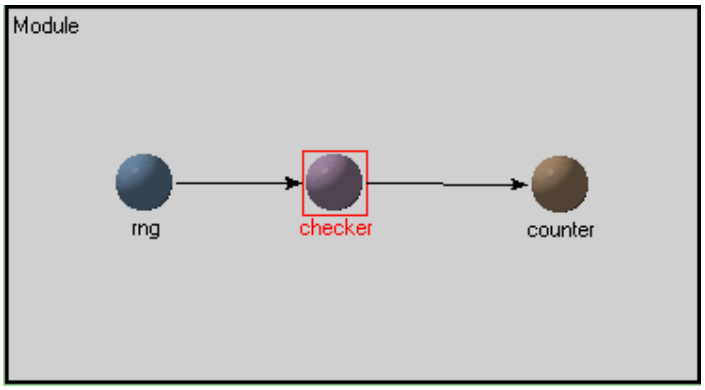


Như vậy trong GNED, hai chế độ giao diện có ảnh hưởng qua lại lẫn nhau. Những thay đổi trong chế độ này cũng ảnh hưởng đến chế độ còn lại.

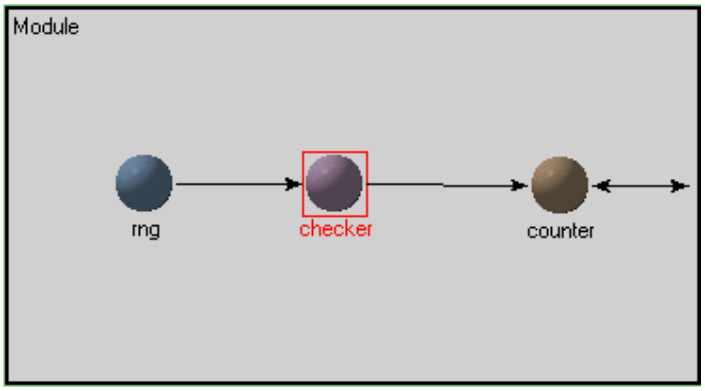
Bây giờ chúng ta sẽ sử dụng các module này để thử nghiệm chức năng của bộ tạo số ngẫu nhiên (Random Number Generator - RNG). Trước tiên, kích phải chuột vào từng module và chọn chức năng Rename để đổi tên cho các module theo thứ tự lần lượt là "rng", "checker" và "counter".

"rng" sẽ sinh ra các số rồi chuyển qua cho "checker". "checker" sẽ tính lại trên mỗi số và chỉ chuyển cho "counter" các số chẵn, các số lẻ sẽ bị bỏ đi. "counter" sẽ ghi lại số lượng số chẵn mà nhận được trong từng phút, và số này sẽ được truyền đi cho các thành phần bên ngoài khi có yêu cầu.

Ta sử dụng công cụ  để tạo kết nối theo mô hình mô tả ở trên. "rng" phải nối với "checker" và "checker" nối với "counter".



"counter" còn có nhiệm vụ nhận và gửi message với các thành phần bên ngoài, do đó ta phải tạo thêm các kết nối cần thiết. Để kết nối một module với bên ngoài, kéo rê chuột từ tâm module tới biên của vùng mầu xám. Để kết nối này là hai chiều, ta tiến hành tạo kết nối theo chiều ngược lại từ biên vào tâm của module.

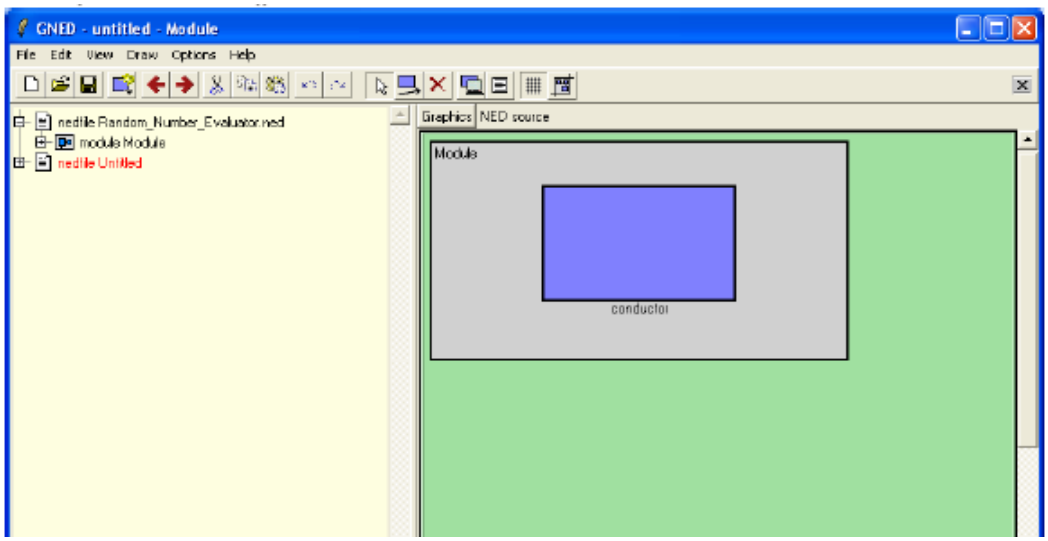


Bây giờ ta đổi tên cho module kết hợp thành "rnd_eval" bằng cách kích phải chuột vào dòng chữ "Module" nằm ở góc trên trái của vùng mầu xám, chọn chức năng Rename trong trình đơn ngữ cảnh tương ứng.

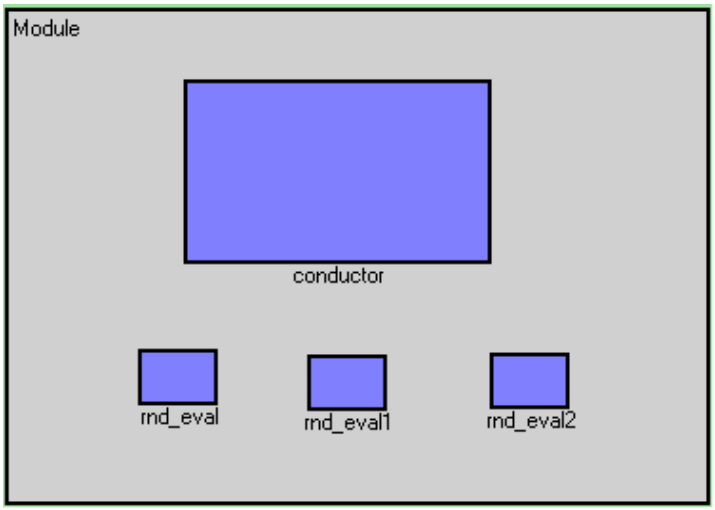
Save file với tên "Random_Number_Evaluator.ned", chúng ta đã có một file NED hoàn chỉnh.

4.3. Làm việc với nhiều file NED - Các chức năng chỉnh sửa nâng cao

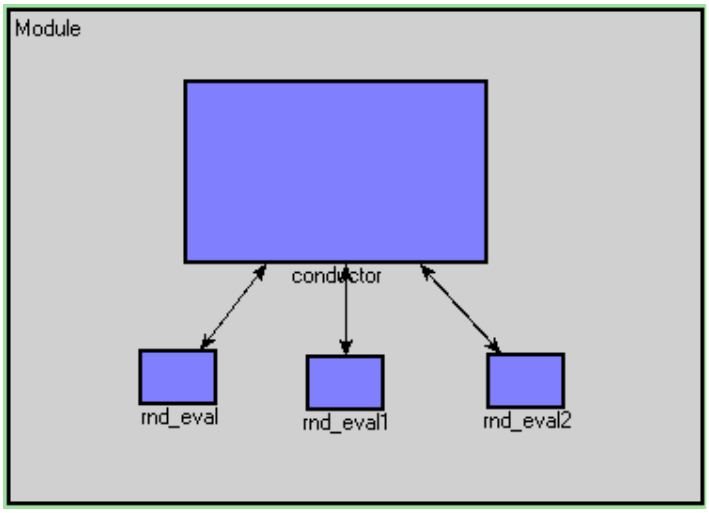
Nhấn nút  để tạo một tài liệu NED mới. Tạo một module con mới có tên là "conductor".



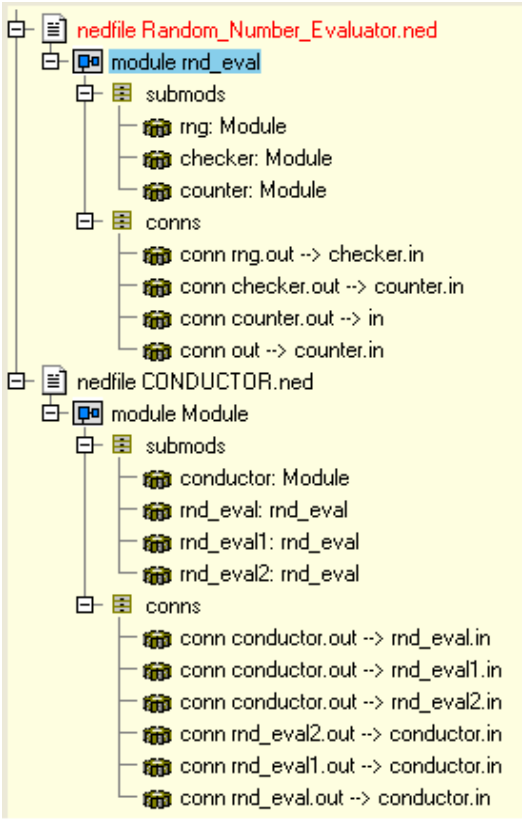
Sau đó kéo rê module "rnd_eval" của file ta vừa tạo ở trên vào module "Module" của file mới ba lần.



Các module "rnd_eval" sẽ lần lượt được đặt tên là "rnd_eval", "rnd_eval1", "rnd_eval2". Sau đó ta tạo kết nối hai chiều giữa module "conductor" với các module "rnd_eval" vừa thêm vào. Module "conductor" sẽ thỉnh thoảng yêu cầu các số liệu của các module "rnd_eval" và các module này sẽ gửi message trả lời.



Quan sát trên Tree View, ta sẽ thấy các kết nối vừa được tự động tạo ra



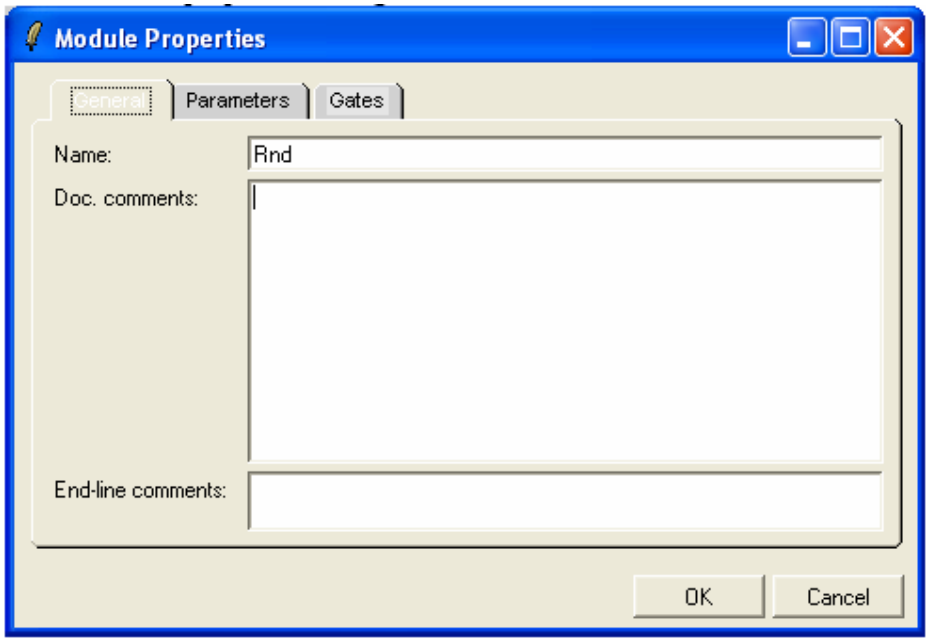
Và file NED cũng có những thay đổi tương ứng

```
module Module
    submodules:
        conductor: Module;
            display: "p=184,96;b=160,96";
        rnd_eval: rnd_eval;
            display: "p=100,205;b=40,28";
        rnd_eval1: rnd_eval;
            display: "p=189,208;b=40,28";
        rnd_eval2: rnd_eval;
            display: "p=285,207;b=40,28";
    connections:
        conductor.out --> rnd_eval.in;
        conductor.out --> rnd_eval1.in;
        conductor.out --> rnd_eval2.in;
        rnd_eval2.out --> conductor.in;
        rnd_eval1.out --> conductor.in;
        rnd_eval.out --> conductor.in;
endmodule
```

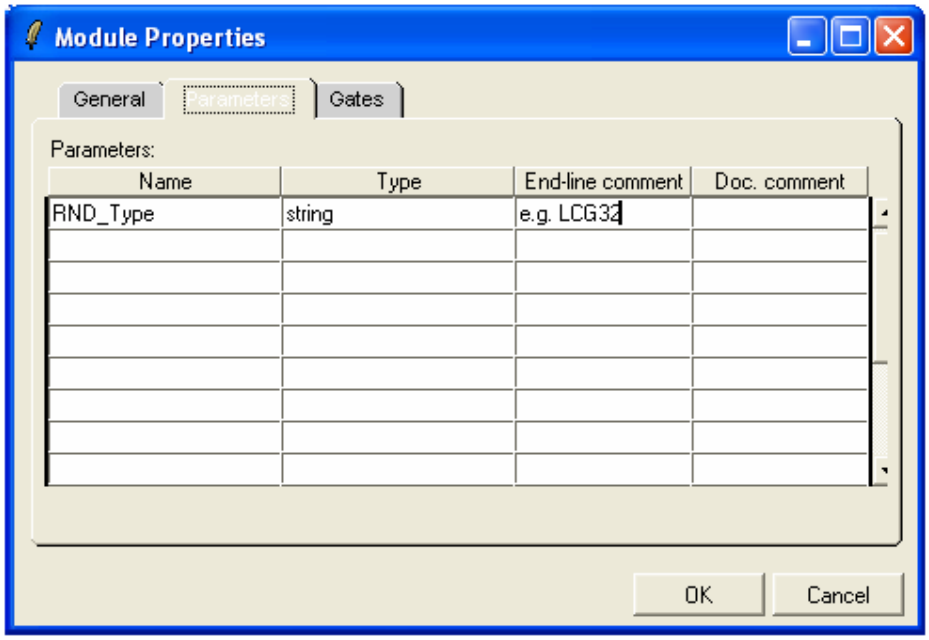
Save file mới này với tên "conductor.ned".

Để thử nghiệm của chúng ta thực sự hoạt động, chúng ta cần phải khai báo "Rnd", "Checker", "Counter" như các module đơn giản.

Bước đầu tiên, chúng ta tạo 3 module đơn giản. Ấn nút  , hoặc sử dụng trình đơn ngữ cảnh trên Tree View để thêm vào một module đơn giản. Cửa sổ Module Properties xuất hiện, chọn tab Reneral và gõ vào ô Name tên của module "Rnd".



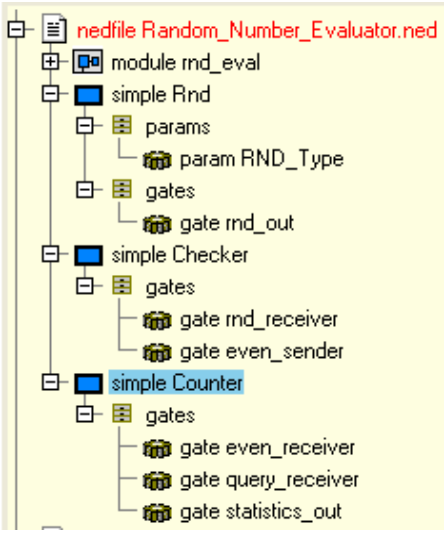
Chọn tab Parameters để định nghĩa các tham số (các tham số này có thể sẽ được các RNG sử dụng), bằng cách này bạn có thể cấu hình lại sau này.



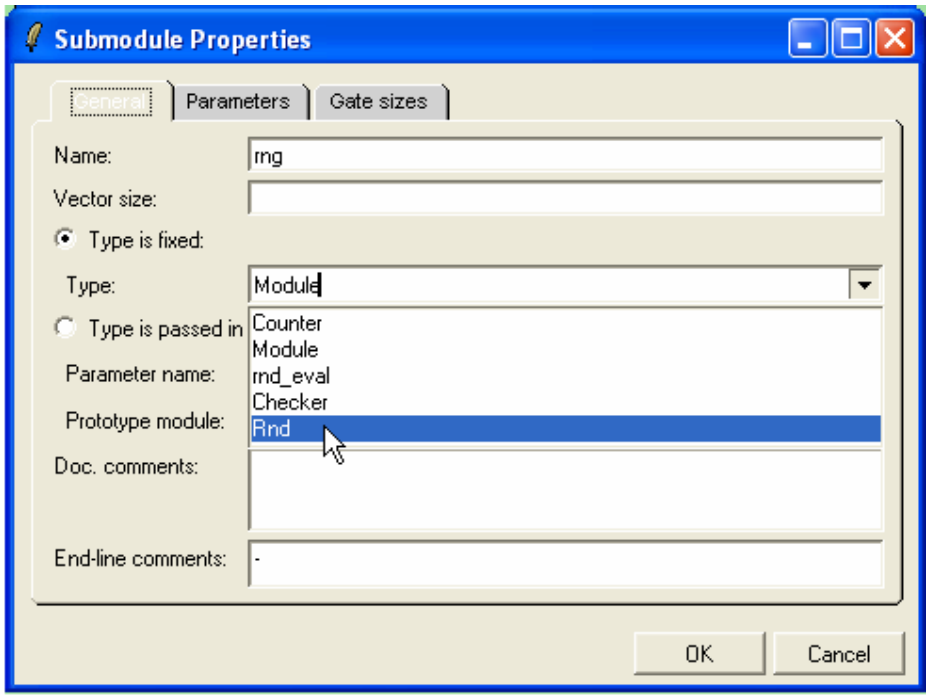
Chọn tab Gates và khai báo một cổng, qua đó các số ngẫu nhiên sinh ra sẽ được truyền đi. Bạn có thể sử dụng trường End-line comment để viết chú thích cho chức năng của cổng.

Bây giờ module Rnd đã sẵn sàng và bạn có thể điều khiển hoạt động của module này thông qua các file tương ứng Rnd.cc và Rnd.h.

Tương tự như trên, ta tiếp tục khai báo các module đơn giản có kiểu "Checker" và "Counter".



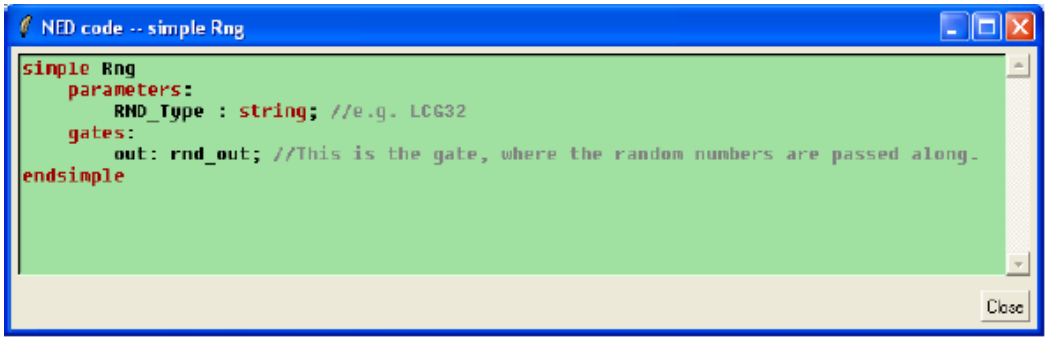
Tiếp theo ta thay đổi kiểu của mỗi module con bên trong module rnd_eval cho phù hợp với các kiểu module đơn giản mới mà ta vừa tạo ra. Kích chuột phải vào tên module "rng", chọn "Properties...", sau đó chọn trong Type kiểu Rnd.



Thay đổi tương tự đối với các module còn lại, sau đó save file lại.

```
module rnd_eval
    submodules:
        rng: Rng; //
            display: "p=83,103;i=ball,#0080ff";
        checker: Checker; //
            display: "p=184,103;i=ball,#ff80ff";
        counter: Counter; //
            display: "p=304,104;i=ball,#ff8000";
    connections:
        rng.out --> checker.in;
        checker.out --> counter.in;
        counter.out --> in;
        out --> counter.in;
endmodule
```

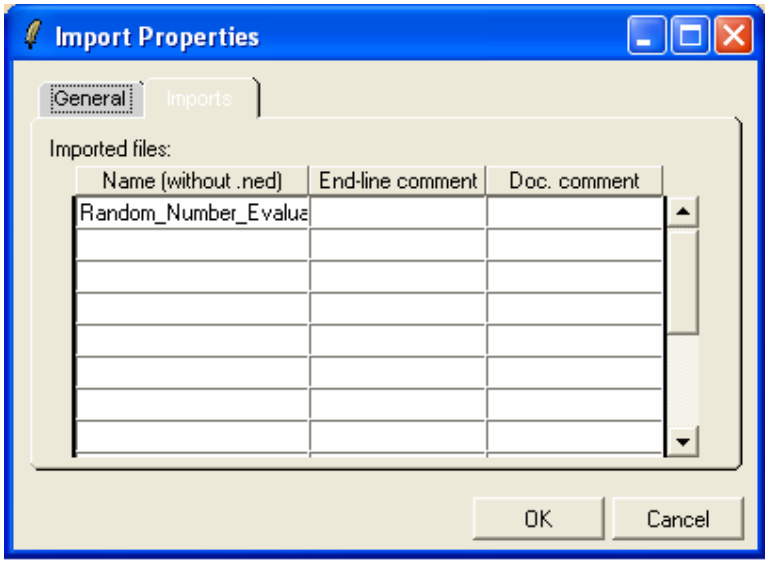
Bạn thấy rằng các module con "rng", "checker" và "counter" đã có các kiểu module tương ứng. Ngoài ra, nếu bạn kích chuột phải vào simple Rnd và chọn "Show NED code...", bạn sẽ thấy những khai báo bạn thực hiện trong cửa sổ Module Properties ở trên được thể hiện dưới dạng mã:



Quay lại với fiel "Conductor.ned", file này sử dụng module "rnd_eval", một module kết hợp được định nghĩa trong file "Random_Number_Evaluator.ned". Do đó bạn

phải import file “Random_Number_Evaluator.ned” để có thể sử dụng module “rnd_eval”.

Ấn nút  và chọn mục Imports



Gõ tên “Random_Number_Evaluator” vào trường Name (chú ý không gõ thêm phần mở rông .ned). Ngoài ra bạn có thể thêm vào các lời chú thích ở hai cột tiếp theo.

Save file conductor.ned.

Bạn đã hoàn thành một mô hình mạng đơn giản. Để chạy được mô hình này, bạn phải triển khai các module đơn giản trong C++ và xây dựng các file .exe.

5. MODULE ĐƠN GIẢN

5.1 Module đơn giản trong OMNeT++

Ta sử dụng C++ để xây dựng các simple module để thực hiện các sự kiện, hay nói khác đi là thực hiện các hoạt động (behaviour) của mô hình.

Các module do người lập trình xây dựng thường là các lớp kế thừa từ lớp cSimpleModule trong thư viện của OMNeT++.

Mỗi module thường chứa các hàm sau:

- void initialize()
- void handleMessage(cMessage *msg)
- void activity()
- void finish()

Hàm khởi tạo initialize()

Trong quá trình khởi tạo, OMNeT++ sẽ xây dựng mạng: nó tạo ra các module đơn và các module kết hợp (compound module). Sau đó kết nối chúng theo các khai báo và định nghĩa trong file NED.

Hàm handleMessage() and activity()

Các hàm này được gọi khi trong quá trình xử lý sự kiện. Như vậy hầu hết hoạt động của hệ thống được mô phỏng sẽ được lập trình trong các hàm này. Hàm handleMessage() sẽ được nhân mô phỏng (simulation kernel) gọi khi module nhận được một gói tin. Và thông thường, ta chủ yếu xây dựng hàm handleMessage() thay vì hàm activity().

Hàm finish()

Hàm finish() được gọi khi quá trình mô phỏng kết thúc thành công. Ngoài ra, một ứng dụng chủ yếu của hàm này còn là thu thập các thống kê về quá trình mô phỏng.

5.2 Các sự kiện trong OMNeT++

OMNeT++ sử dụng các message để biểu diễn các sự kiện. Mỗi sự kiện được biểu diễn bởi một đối tượng của lớp cMessage hoặc lớp con của lớp này.

5.3 Mô hình hoá hoạt động truyền gói tin

Tương tự như đã trình bầy trong phần 2.1.4, Một kết nối có thể có ba tham số đặc trưng. Những tham số này rất thuận tiện cho các mô hình mô phỏng mạng thông tin nhưng không hữu dụng lắm cho các kiểu mô hình khác. Ba tham số này bao gồm:

- Độ trễ đường truyền (propagation delay) tính bằng s - giây.

- Tỉ số lỗi bit, được tính bằng số lỗi/bit.
- Tỉ số dữ liệu, được tính bằng số bit/s.

Các tham số này là tuỳ chọn. Giá trị của các tham số này là khác nhau trên từng kết nối, phụ thuộc vào kiểu của liên kết (hay còn gọi là kiểu của kênh truyền - channel type).

5.4 Khai báo kiểu module đơn giản

5.4.1 Tổng quan

Một module viết bằng C++ bao gồm:

- Khai báo lớp module: lớp do người lập trình xây dựng là lớp con của lớp cSimpleModule.
- Đăng ký kiểu của module (module type registration): Define_Module() hoặc Define_Module_Like() macro.
- Triển khai cụ thể lớp module.

Xét VD sau:

```
// file: swp.cc
#include <omnetpp.h>

// module class declaration:
class SlidingWindowProtocol : public cSimpleModule
{
   Module_Class_Members(SlidingWindowProtocol,cSimpleModule,0)
   virtual void handleMessage(cMessage *msg);
};

// module type registration:
Define_Module( SlidingWindowProtocol );

// implementation of the module class:
void SlidingWindowProtocol::handleMessage(cMessage *msg)
{
  ...
}
```

Module trên được khai báo trong file NED như sau:

```
// file: swp.ned
```

```
simple SlidingWindowProtocol
    parameters:
        windowSize: numeric const;
    gates:
        in: fromNetw, fromHigherLayer;
        out: toNetw, toHigherLayer;
endsimple
```

5.4.2 Đăng ký kiểu module

Trong VD trên chứa câu lệnh:

```
Define_Module(SlidingWindowProtocol);
```

Dòng lệnh này sẽ khiến cho OMNeT++ biết rằng ta muốn dùng lớp SlidingWindowProtocol dưới dạng 1 module đơn. Đồng thời OMNeT++ framework sẽ tìm file **NED** có cùng tên chứa khai báo về module này

(simple SlidingWindowProtocol
...
endsimple)

để xác định các cổng và các tham số mà module này cần có.

5.5 Xây dựng hàm cho Module

5.5.1 Hàm handleMessage()

Hàm handleMessage() được gọi khi mỗi message đến module. Khi đó hàm này sẽ xử lý gói tin và trả lại kết quả ngay lập tức.

Chú ý rằng các module hàm handleMessage() KHÔNG được tự động gọi, mà phải nhận được gói tin từ module khác. Muốn ta phải thêm các self-message từ hàm khởi tạo initialize() thì hàm handleMessage() sẽ bắt đầu làm việc mà không cần phải nhận gói tin từ module khác.

Để sử dụng hàm handleMessage() trong một module, ta phải xác định kích thước của zero stack size cho module đó. Bởi lẽ kích thước của zero stack sẽ khiến OMNeT++ biết ta muốn sử dụng hàm handleMessage() hay activity().

Một số hàm thông dụng mà ta có thể sử dụng trong hàm handleMessage():

- Các hàm send(): gửi gói tin tới các module khác.
- Hàm scheduleAt(): để định kỳ một sự kiện (thường là module tự gửi gói tin cho chính nó)
- Hàm cancelEvent(): hủy bỏ định kỳ một sự kiện nhờ hàm scheduleAt()

Chú ý rằng các hàm receive() và wait() không được sử dụng trong việc xây dựng hàm handleMessage(), mà chỉ dùng khi ta muốn xây dựng hàm activity().

- **VD 1:**

```
class FooProtocol : public cSimpleModule
{
  protected:
    // state variables
    // ...

    virtual void processMsgFromHigherLayer(cMessage *packet);
    virtual void processMsgFromLowerLayer(FooPacket *packet);
    virtual void processTimer(cMessage *timer);

  public:
    Module_Class_Members(FooProtocol, cSimpleModule, 0);
    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);
}
```

5.5.2 Hàm activity()

Các hàm quan trọng mà ta có thể gọi trong hàm này bao gồm:

- receive()
- wait()
- send()
- scheduleAt()
- cancelEvent()
- end()

5.5.3 Hàm initialize() và finish()

Hàm initialize(): khởi tạo các giá trị cần thiết cho quá trình mô phỏng

Hàm finish(): hàm này được sử dụng để ghi lại các thông số trạng thái cần thiết khi quá trình mô phỏng kết thúc.

5.6 Gửi và nhận các message

5.6.1 Gửi các message

Sau khi tạo ra các gói tin, ta có thể gửi nó thông qua một cổng vào/ra nhờ hàm send() với cú pháp như sau:

```
send(cMessage *msg, const char *gateName, int index=0);
send(cMessage *msg, int gateId);
send(cMessage *msg, cGate *gate);
```

Ví dụ:

```
send(msg, "outGate");
send(msg, "outGates", i); // send via outGates[i]
```

Đoạn mã sau sẽ tạo ra và gửi các gói tin sau mỗi 5 giây.

```
int outGateId = findGate("outGate");
while(true)
{
  send(new cMessage("packet"), outGateId);
  wait(5);
}
```

5.6.2 Broadcasts

Khi ta muốn cùng một gói tin tới nhiều nút đích đồng thời, thì dùng phương pháp tạo ra nhiều bản sao của gói tin và gửi chúng đi.

Ví dụ:

```
for (int i=0; i<n; i++)
{
    cMessage *copy = (cMessage *) msg->dup();
    send(copy, "out", i);
}
delete msg;
```

5.6.3 Gửi có độ trễ (Delayed sending)

```
wait( someDelay );
send( msg, "outgate" );
```

Cú pháp cũng tương tự như trên, chỉ thêm tham số thời gian trễ

```
sendDelayed(cMessage *msg, double delay, const char *gateName, int index);
sendDelayed(cMessage *msg, double delay, int gateId);
sendDelayed(cMessage *msg, double delay, cGate *gate);
```

Ví dụ:

```
sendDelayed(msg, 0.005, "outGate");
```

5.6.4 Gửi trực tiếp message

Sử dụng hàm sendDirect() để gửi trực tiếp gói tin từ module này tới module kia mà không cần quan tâm đến thông qua cổng nào.

```
sendDirect(cMessage *msg, double delay, cModule *mod, int gateId)
sendDirect(cMessage *msg, double delay, cModule *mod, const char *gateName,
int index=-1)
sendDirect(cMessage *msg, double delay, cGate *gate)
```

Ví dụ

```
cModule *destinationModule = parentModule()->submodule("node2");
double delay = truncnormal(0.005, 0.0001);
sendDirect(new cMessage("packet"), delay, destinationModule, "inputGate");
```

5.6.5 Gửi định kỳ

```
scheduleAt(absoluteTime, msg);
scheduleAt(simtime()+delta, msg);
```

5.7 Truy nhập các cổng và kết nối

5.7.1 Đối tượng cổng (gate object)

Module cổng là một đối tượng của lớp cGate. Hàm gate() sẽ trả về 1 con trỏ tới đối tượng cGate. Và muốn truy cập vào từng thành phần của cổng đó, ta thực hiện chồng hàm

```
cGate *outgate = gate("out");
cGate *outvec5gate = gate("outvec",5);
```

Gate ID

Các module cổng được lưu trữ trong một mảng. Vi trí của cổng trong mảng đó ội là *gate ID*. Để xác định gate ID, ta dùng hàm id() hoặc hàm findGate()

```
int id = outgate->id();
or:
int id1 = findGate("out");
int id2 = findGate("outvect",5);
```

Nhờ đó, có thể gửi và nhận gói tin thông qua tham số là gate ID. Thông thường thì việc sử dụng gate ID sẽ nhanh hơn là dùng tên cổng.

5.7.2 Các tham số kết nối

Các thông số thông số cơ bản của đường truyền: độ trễ, tỉ lệ bit lỗi, tốc độ truyền được biểu diễn thông qua đối tượng channel.

```
cChannel *chan = outgate->channel();
```

```
cBasicChannel *chan = check_and_cast<cBasicChannel *>(outgate->channel());
double d = chan->delay();
double e = chan->error();
double r = chan->datarate();
```

5.8 Tự động tạo module

Trong một số tình huống, ta cần phải tự động tạo và hủy các module. Chẳng hạn khi mô phỏng một mạng di động, ta cần tạo một module mới khi người dùng tiến vào vùng kết nối và hủy module này khi người đó ra khỏi vùng kết nối.

Quá trình trên gồm 5 bước:

1. Tìm loại module.
2. Tạo module
3. Thiết lập các tham số và kích thước cổng (nếu cần)
4. Gọi hàm xây dựng (build out) các module con và hoàn thành module chính.
5. Gọi hàm tạo các gói tin chủ động (activation message) cho mỗi module đơn.

Ví dụ:

```
// find factory object
cModuleType *moduleType = findModuleType("WirelessNode");

// create (possibly compound) module and build its submodules (if any)
```

```
cModule *module = moduleType->create("node", this);
module->buildInside();

// create activation message
module->scheduleStart( simTime() );
```

Hủy module

```
module->deleteModule();
```

Tạo các kết nối

```
srcGate->connectTo(destGate);
```

Tạo 2 module và kết nối chúng với nhau:

```
cModuleType *moduleType = findModuleType("TicToc");
cModule *a = modtype->createScheduleInit("a",this);
cModule *b = modtype->createScheduleInit("b",this);

a->gate("out")->connectTo(b->gate("in"));
b->gate("out")->connectTo(a->gate("in"));
```

Hủy kết nối

```
srcGate->disconnect();
```

6. MESSAGE

6.1. Message và Packet

6.1.1. Lớp cMessage

cMessage là một lớp trung tâm của OMNeT++. Đối tượng của lớp cMessage và các lớp con của nó có thể mô hình hoá được rất nhiều đối tượng như các message, các gói tin (packet), frame, cell, bit, các tín hiệu truyền trong mạng, các thực thể truyền trong một hệ thống...

Thuộc tính

Một đối tượng của lớp cMessage có một số thuộc tính, một số được sử dụng bởi phần nhân mô phỏng, một số khác được cung cấp cho người lập trình.

- Tên - name: thuộc tính là một chuỗi (const char *) mà người lập trình có thể sử dụng tuỳ ý. Tên của message xuất hiện ở rất nhiều nơi trong Tkenv và nên được chọn có ý nghĩa. Thuộc tính này kế thừa từ lớp cObject.
- Kiểu message - message kind: thuộc tính này chứa thông tin về kiểu của message.
- Độ dài - length (được tính theo bit): được sử dụng để tính độ trễ khi message được truyền thông qua một kết nối có tốc độ truyền dữ liệu được gán giá trị xác định.
- Cờ bit lỗi - bit error flag: thuộc tính này được thiết lập bằng true bởi phần nhân mô phỏng với xác suất bằng 1-(1-ber)length khi message được gửi thông qua một kết nối có tốc độ truyền dữ liệu xác định (ber).
- Quyền ưu tiên - priority: được sử dụng bởi phần nhân mô phỏng để sắp xếp các message trong danh sách hàng đợi (message queue - FES) có cùng thời gian tới.
- Mốc thời gian - time stamp: thuộc tính này cho phép người sử dụng đánh dấu thời gian ví dụ như đánh dấu thời điểm message được xếp vào hàng đợi hoặc được gửi lại.
- Các thuộc tính khác và các thành phần dữ liệu giúp cho người lập trình làm việc dễ dàng hơn như: danh sách tham số (parameter list), message đóng gói (encapsulated message), thông tin điều khiển (control info) và con trỏ ngữ cảnh (context pointer.
- Một số các thuộc tính chỉ đọc (read-only attribute) lưu giữ các thông tin về việc gửi message, các thông tin về các module, cổng nguồn và đích, thời gian gửi và thời gian tới của các message. Hầu hết các thuộc tính này đều được sử dụng bởi phần nhân mô phỏng khi các message nằm trong FES, tuy nhiên khi các module nhận được message, các thông tin này vẫn còn tồn tại.

Cách sử dụng

Hàm khởi tạo của lớp cMessage có thể nhận một vài đối số. Thông thường, một đối tượng của lớp cMessage sẽ nhận vào hai đối số là tên (kiểu string) và kiểu message (kiểu int):

cMessage *msg = new cMessage("MessageName", msgKind);

Tất cả các đối số đều là tuỳ chọn, do đó khai báo một đối tượng như sau cũng hợp lệ

cMessage *msg = new cMessage();

hay

cMessage *msg = new cMessage("MessageName");

Khi không có đối số, mặc định đối tượng mới tạo ra có tên là "" và kiểu là 0. Hàm tạo của lớp cMessage có thể nhận vào nhiều đối số hơn (length, priority, bit error flag), tuy nhiên để đặt giá trị cho các thuộc tính ta cũng không nhất thiết phải sử dụng hàm tạo. Ta có thể sử dụng hàm set...() để gán giá trị cho từng thuộc tính.

msg->setKind(kind);

msg->setLength(length);

msg->setPriority(priority);

msg->setBitError(err);

msg->setTimestamp();

msg->setTimestamp(simtime);

Ngoài ra ta có thể sử dụng các hàm sau để lấy giá trị của các tham số:

int k = msg->kind();

int p = msg->priority();

int l = msg->length();

bool b = msg->hasBitError();

simtime_t t = msg->timestamp();

Nhân đôi message

Ta có thể thực hiện sao chép một message bằng cách:

cMessage *copy = (cMessage *) msg->dup();

hoặc

cMessage *copy = new cMessage(*msg);

Cách này có thể áp dụng với bất kỳ một đối tượng nào trong OMNeT++. Message mới được tạo là một bản copy chính xác của message cũ, bao gồm cả các tham số...

6.1.2. Self-Message

Sử dụng self-message

Các message thường được sử dụng để mô tả các sự kiện xảy ra bên trong của một module. Trong một số trường hợp, message có thể coi như một bộ định thời dùng để xác định thời điểm diễn ra một sự kiện nào đó. Những message sử dụng trong những

trường hợp như vậy được gọi là self-message. Tuy nhiên self-message vẫn là message bình thường, là một đối tượng của lớp cMessage hoặc một lớp con kế thừa từ nó.

Khi một message được phân đến một module bởi phần nhân mô phỏng, bạn có thể gọi hàm isSelfMessage() để kiểm tra xem nó có phải là một self-message hay không; nói một cách khác là để kiểm tra xem message nhận được có phải là một scheduled message (các message dùng để định thời điểm diễn ra một sự kiện trong module) hay là các message được gửi bởi một hàm send...() nào đó. Ngoài ra người sử dụng có thể sử dụng hàm isScheduled() để kiểm tra, hàm này sẽ trả về true nếu message nhận được là một scheduled message (những message được xác định bởi hàm scheduleAt()). Một scheduled message cũng có thể bị huỷ bỏ bởi hàm cancelEvent().

bool isSelfMessage();

bool isScheduled();

Các hàm sau trả về thời gian tạo, thiết lập và thời gian tới của một message. simtime_t creationTime()

simtime_t sendingTime();

simtime_t arrivalTime();

Khi một self-message được thiết lập, thời gian tới của message sẽ là thời gian nó sẽ được chuyển tới module cần thiết.

Con trỏ ngữ cảnh (Context Pointer)

Xét hai hàm setContextPointer() và contextPointer():

Hàm setContextPointer() nhận một con trỏ ngữ cảnh (kiểu void) làm đối số để thiết lập ngữ cảnh cho message.

Hàm contextPointer() trả về một con trỏ kiểu void, chứa ngữ cảnh của message tương ứng.

```
void *context =...;

msg->setContextPointer( context );

void *context2 = msg->contextPointer();
```

Người lập trình có thể sử dụng con trỏ ngữ cảnh cho nhiều mục đích và phần nhân mô phỏng không can thiệp đến con trỏ này. Tuy nhiên trên thực tế, con trỏ ngữ cảnh thường được sử dụng khi một module thiết lập một vài self-message (bộ định thời), module sẽ cần phải xác định được khi nào một self-message quay lại module, hay nói một cách khác nó cần phải xác định được khi nào bộ định thời hoạt động và phải làm gì sau đó. Khi đó con trỏ ngữ cảnh sẽ được tạo ra để trỏ tới một cấu trúc dữ liệu của module, mang đầy đủ thông tin "ngữ cảnh" về sự kiện sắp diễn ra.

6.1.3. Mô hình hoá gói tin

Cổng nhận và thời gian tới của một message

Các hàm chỉ ra vị trí nhận và gửi của một message:

int senderModuleId();

int senderGateId();

int arrivalModuleId();

int arrivalGateId();

Hai hàm được sử dụng để kết hợp module id với gate id thành một con trỏ đối tượng cổng (gate object pointer):

cGate *senderGate();

cGate *arrivalGate();

Các hàm dùng để kiểm tra xem message gửi đến được nhận vào từ cổng nào thông qua id hoặc tên + chỉ số của cổng:

bool arrivedOn(int id);

bool arrivedOn(const char *gname, int gindex=0);

Các hàm trả lại thời gian tạo message, thời gian gửi lần cuối cùng và thời gian tới của message:

simtime_t creationTime()

simtime_t sendingTime();

simtime_t arrivalTime();

Thông tin điều khiển

Một trong những lĩnh vực ứng dụng chủ yếu của OMNeT++ là mạng thông tin. Trong lĩnh vực này, các lớp giao thức thường được triển khai như những module làm nhiệm trao đổi các gói tin. Lớp cMessage cũng cung cấp một lớp con của nó để khai báo các gói tin.

Tuy nhiên, việc thông tin giữa các lớp giao thức cần phải có những thông tin phụ được gắn kèm cùng với gói tin. Lấy ví dụ, khi lớp TCP gửi một gói tin xuống lớp IP, gói tin cần phải có địa chỉ IP nhận và một số tham số khác nữa. Khi lớp IP chấp nhận gói tin từ lớp TCP (đọc thông tin địa chỉ IP nhận trong phần header của gói tin), nó sẽ chuyển ngược các thông tin cần thiết lên cho lớp TCP, ít nhất là địa chỉ IP nguồn gửi gói tin.

Các thông tin thêm vào được coi là các đối tượng thông tin điều khiển (control info object) trong OMNeT++. Các đối tượng thông tin điều khiển này là đối tượng của lớp con kế thừa từ lớp cPolymorphic (một lớp không có thành phần dữ liệu), và được gắn kèm vào các gói tin. Các hàm có thể sử dụng cho mục đích này:

void setControlInfo(cPolymorphic *controlInfo);

cPolymorphic *controlInfo();

cPolymorphic *removeControlInfo();

Xác định giao thức

Trong các mô hình giao thức của OMNeT++, kiểu giao thức thường được đại diện bởi cấu trúc của các gói tin sử dụng trong giao thức và được thể hiện như một lớp message. Ví dụ như lớp IPv6Datagram tương ứng với các datagram của IPv6, hay lớp EthernetFrame tương ứng với các frame của Ethernet. Các kiểu đơn vị dữ liệu của giao thức (PDU - Protocol Data Unit) thường được thể hiện như một trường trong lớp message.

Trong C++, toán tử dynamic_cast có thể được sử dụng để kiểm tra xem một đối tượng message có thuộc một kiểu giao thức xác định nào đó hay không.

```
cMessage *msg = receive();
if (dynamic_cast<IPv6Datagram *>(msg) != NULL)
{
IPv6Datagram *datagram = (IPv6Datagram *)msg;
...
}
```

6.1.4. Đóng gói (Encapsulation)

Đóng gói gói tin

Thực sự cần thiết phải đóng gói một message khi bạn tiến hành mô phỏng các lớp giao thức của một mạng máy tính. Các tốt nhất để đóng gói một message là thêm vào message một danh sách các tham số đặc biệt. OMNeT++ cung cấp cho người sử dụng hàm encapsulate() để đóng gói các message. Kích thước (chiều dài) của các message sẽ tăng lên một phần bằng kích thước của phần thông tin thêm vào.

```
cMessage *userdata = new cMessage("userdata");
userdata->setLength(8*2000);
cMessage *tcpseg = new cMessage("tcp");
tcpseg->setLength(8*24);
tcpseg->encapsulate(userdata);
ev << tcpseg->length() << endl; // --> 8*2024 = 16192
```

Một message chỉ có thể mang một phần thông tin thêm. Điều này có nghĩa là nếu hàm encapsulate() được gọi lần thứ hai, nó sẽ sinh ra lỗi. Ngoài ra lỗi cũng phát sinh nếu message được đóng gói không thuộc một module nào.

Bạn có thể lấy lại phần thông tin thêm vào bằng hàm decapsulate()

```
cMessage *userdata = tcpseg->decapsulate();
```

Hàm decapsualate() sẽ làm chiều dài của message (trừ trường hợp phần thêm vào có chiều dài bằng 0). Nếu chiều dài của message sau khi thực hiện lệnh trở thành số âm, sẽ xuất hiện lỗi.

Hàm encapsulatedMsg() trả về một con trỏ trỏ tới message được đóng gói, nếu giá trị trả về bằng NULL có nghĩa là không có message nào được đóng gói.

Đóng gói nhiều message

Lớp cMessage không trực tiếp hỗ trợ việc thêm nhiều hơn một message vào một đối tượng message (message object), nhưng bạn có thể tạo một lớp con của lớp cMessage và thêm vào các chức năng cần thiết.

Bạn cũng có thể đặt nhiều message trong một mảng có kích thước cố định hoặc một mảng cấp phát động, hoặc bạn có thể sử dụng một số lớp STL như std::vector hay std::list. Tuy nhiên có một điểm mà bạn cần lưu ý khi sử dụng các lớp này đó là: lớp message của bạn phải chiếm quyền sở hữu của các message được thêm vào, và phải

giải phóng chúng khi chúng bị xóa khỏi danh sách. Những việc này được thực hiện qua hai hàm take() và drop(). Ví dụ để thêm vào lớp một std::list được gọi là messages chứa danh sách các con trỏ message, bạn nên thêm đoạn mã sau:

```
void MessageBundleMessage::insertMessage(cMessage *msg)
{
take(msg); // take ownership
messages.push_back(msg); // store pointer
}
void MessageBundleMessage::removeMessage(cMessage *msg)
{
messages.remove(msg); // remove pointer
drop(msg); // release ownership
}
```

Bạn cũng cần phải thêm vào một hàm operator=() để đảm bảo các đối tượng message của bạn có thể được sao chép hoặc nhân đôi (đây là những điều rất cần thiết trong quá trình mô phỏng).

6.1.5. Thêm đối tượng và tham số

Thêm đối tượng

Lớp cMessage có đối tượng cArray có thể chứa các đối tượng khác. Tuy nhiên chỉ các đối tượng kế thừa từ lớp cObject (hầu hết các lớp trong OMNeT++ đều kế thừa từ lớp này) mới có thể được thêm vào các message. Các hàm addObject(), hasObject(), removeObject() nhận tên của các đối tượng như các khoá để truy nhập mảng. Ví dụ:

```
cLongHistogram *pklenDistr = new cLongHistogram("pklenDistr");
msg->addObject( pklenDistr );
...
if (msg->hasObject("pklenDistr"))
{
cLongHistogram *pklenDistr =
(cLongHistogram *) msg->getObject("pklenDistr");
...
}
```

Bạn phải cẩn thận khi thêm một đối tượng vào message, tránh để xẩy ra tình trạng xung đột giữa các đối tượng bị trùng tên. Nếu bạn không gắn kèm một đối tượng nào vào message anh không gọi hàm parList(), đối tượng cArray sẽ không được tạo.

Bạn cũng có thể thêm vào message các đối tượng không kế thừa từ cObject (non-cObject object) bằng cách sử dụng con trỏ của lớp cPar. Ví dụ:

```
struct conn_t *conn = new conn_t; // conn_t is a C struct
```

```
msg->addPar("conn") = (void *) conn;
msg->par("conn").configPointer(NULL,NULL,sizeof(struct conn_t));
```

Thêm tham số

Phương pháp tốt nhất để mở rộng các message với những trường dữ liệu mới là định nghĩa các message (xem phần 5.2).

Tuy nhiên ta có thể sử dụng một phương pháp khác (không được khuyến khích) để thêm các trường dữ liệu mới cho message thông qua các đối tượng cPar. Nhược điểm của phương pháp này là tốn bộ nhớ và thời gian thực hiện chậm. Các đối tượng của cPar thường có kích thước lớn và khá phức tạp. Mặt khác khi sử dụng các đối tượng cPar cũng rất dễ sinh ra lỗi bởi các đối tượng này phải được thêm vào động và độc lập đối với mỗi đối tượng message.

Tuy nhiên nếu bạn vẫn cần sử dụng cPar, nó sẽ cung cấp cho bạn một số hàm cơ bản. Hàm addPar() được dùng để thêm một tham số mới cho message. Hàm hasPar() kiểm tra xem một message có các tham số hay không. Các tham số của message có thể được truy nhập thông qua chỉ số của mảng tham số. Hàm findPar() trả về chỉ số của một tham số và trả về -1 nếu tham số đó không tồn tại. Các tham số cũng có thể được truy nhập bằng cách viết chồng hàm par(). Ví dụ:

```
msg->addPar("destAddr");
msg->par("destAddr") = 168;
...
long destAddr = msg->par("destAddr");
```

6.2. Định nghĩa message

6.2.1. Giới thiệu

Trong thực tế, bạn sẽ phải thêm rất nhiều trường vào lớp cMessage để làm cho nó dễ dùng hơn. Lấy ví dụ, nếu bạn mô hình hoá các gói tin trong một mạng thông tin, bạn cần có cách để lưu phần header của giao thức trong các đối tượng message. Một cách tự nhiên, chúng ta thấy rằng thư viện mô phỏng của OMNeT++ được viết trên ngôn ngữ C++, do đó để thêm các trường mới vào lớp cMessage ta có thể tạo các lớp con kế thừa từ lớp cMessage và thêm các trường vào như những thành phần riêng của lớp con. Tuy nhiên, do mỗi trường mà bạn thêm vào đều cần ít nhất 3 thành phần (dữ liệu thành phần riêng, hàm set() để thiết lập giá trị và hàm get() để trả về giá trị) và lớp mới cần phải được tích hợp với nền tảng mô phỏng nên việc sử dụng C++ thực sự là một công việc buồn tẻ và mất thời gian.

OMNeT++ cung cấp cho người sử dụng một phương pháp làm việc hiệu quả hơn, đó là định nghĩa các message. Những định nghĩa này sử dụng một cú pháp rất ngắn gọn để mô tả nội dung của các message. Mã C++ sẽ tự động sinh ra dựa vào những định nghĩa này và bạn hoàn toàn có khả năng sửa lại những đoạn code cho thích hợp về ý tưởng của bạn.

Lớp message đầu tiên

Ta xét một ví dụ đơn giản. Giả sử rằng bạn cần các đối tượng message phải có thêm địa chỉ của nguồn, đích và bộ đếm bước truyền, bạn có thể viết một file mypacket.msg như sau:

```
message MyPacket
{
fields:
int   srcAddress;
int destAddress;
int hops = 32;
};
```

Nếu bạn biên dịch file mypacket.msg, trình biên dịch sẽ tự động sinh ra hai file C++ tương ứng có tên là mypacket_m.h và file mypacket_m.cc. File mypacket_m.h chứa các khai báo của lớp MyPacket (lớp C++ tương ứng với định nghĩa message trong file mypacket.msg) và bạn có thể đặt file này vào trong mã C++ để điều khiển hoạt động của đối tượng MyPacket.

File mypacket_m.h sẽ chứa các khai báo lớp như sau:

```
class MyPacket : public cMessage
{
...
virtual int getSrcAddress() const;
virtual void setSrcAddress(int srcAddress);
...
};
```

Do đó trong file C++ bạn có thể sử dụng lớp MyPacket như sau:

```
#include "mypacket_m.h"
...
MyPacket *pkt = new MyPacket("pkt");
pkt->setSrcAddress( localAddr );
...
```

File mypacket_m.cc chứa các triển khai của lớp MyPacket cho phép bạn kiểm tra những cấu trúc dữ liệu trong giao diện của Tkenv (Tkenv GUI). File mypacket_m.cc nên được biên dịch và thiết lập liên kết với mô hình mô phỏng của bạn (nếu bạn sử dụng công cụ opp_makemake để tạo các makefiles, thì các công việc liên quan đến file .cc sẽ tự động được thực hiện).

Khái niệm - định nghĩa message

Có nhiều ý kiến không rõ ràng về mục đích cũng như khái niệm về định nghĩa message. Tuy nhiên chúng ta phải xác định rõ ràng rằng, định nghĩa message không phải là:

... sự cố gắng mô phỏng các chức năng của C++ nhưng với một cú pháp khác. Chỉ đơn giản việc định nghĩa message chỉ là xác định các dữ liệu (hay xác định một giao tiếp để truy nhập tới dữ liệu) chứ không phải là bất kỳ một kiểu thuộc tính nào.

... một công cụ sinh mã. Điều này chỉ đúng với việc định nghĩa nội dung của message và các cấu trúc dữ liệu mà bạn sử dụng trong message. Việc định nghĩa các hàm để kiểm soát hoạt động của message không được hỗ trợ. Hơn nữa cả việc sử dụng cú pháp này để sinh ra các lớp và các cấu trúc bên trong của các module đơn giản cũng không được khuyến khích.

6.2.2. Sử dụng enum

enum {...} sử dụng trong khai định nghĩa message sẽ được chuyển thành kiểu enum thực sự trong C++. Đây là một đối tượng dùng để chứa các giá trị text đại diện cho các hằng số. Việc sử dụng enum cho phép hiển thị tên dưới dạng biểu tượng trong Tkenv.

Ví dụ:

```
enum ProtocolTypes
{
IP = 1;
TCP = 2;
};
```

Các giá trị trong enum phải là duy nhất.

6.2.3. Khởi tạo cho một message

Cách khởi tạo cơ bản

Bạn có thể mô tả một message theo cú pháp sau:

```
message FooPacket
{
fields:
int sourceAddress;
int destAddress;
bool hasPayload;
};
```

Trình biên dịch sẽ dịch đoạn mô tả trên thành một file C++ với một lớp có tên là FooPacket. FooPacket này sẽ là một lớp con của lớp cMessage. Đối với mỗi trường trong đoạn khai báo trên, trong lớp C++ tương ứng cũng sẽ có một thành phần dữ liệu riêng, một hàm setter và một hàm getter. Do đó FooPacket sẽ có những hàm sau:

```
virtual int getSourceAddress() const;
virtual void setSourceAddress(int sourceAddress);
virtual int getDestAddress() const;
virtual void setDestAddress(int destAddress);
```

virtual bool getHasPayload() const;

virtual void setHasPayload(bool hasPayload);

Chú ý là tất cả các hàm trên đều có kiểu là vitual, tức là bạn có khả năng thực hiện chồng hàm ở các lớp con.

Hai hàm tạo cũng được sinh ra: một để nhập tên đối tượng và kiểu message và một là hàm tạo sao chép (tạo một đối tượng mới là bản sao của đối tượng cũ).

FooPacket(const char *name=NULL, int kind=0);

FooPacket(const FooPacket& other);

Ngoài ra trình biên dịch cũng tự động sinh ra trong lớp các hàm như operator=() và dup() (các hàm dùng để sao chép và nhân bản đối tượng).

Bạn có thể sử dụng các kiểu dữ liệu dưới đây để khai báo cho các trường trong định nghĩa message:

bool

char, unsigned char

short, unsigned short

int, unsigned int

long, unsigned long

double

Giá trị khởi tạo của các trường mặc định bằng 0.

Giá trị khởi tạo

Bạn có thể khởi tạo giá trị cho các trường trong một message theo cú pháp sau:

message FooPacket

{

fields:

int sourceAddress = 0;

int destAddress = 0;

bool hasPayload = false;

};

Phần mã khởi tạo trong đoạn khai báo trên sẽ được thay thế bởi các hàm tạo trong các lớp C++.

Khai báo kiểu enum

Bạn có thể khai báo các trường kiểu int (hay các kiểu số nguyên khác) nhận giá trị trong một enum. Trong trường hợp sử dụng enum, trình biên dịch có thể sinh mã cho phép Tkenv hiển thị giá trị của trường dưới dạng các biểu tượng.

Ví dụ:

message FooPacket

{

fields:

int payloadType enum(PayloadTypes);

};

Kiểu enum phải được khai báo riêng rẽ trong file .msg.

Mảng kích thước cố định

Có thể sử dụng các mảng có kích thước cố định:

message FooPacket

{

fields:

long route[4];

};

Trong trường hợp này các hàm getter và setter sẽ có thêm một tham số phụ là chỉ số của mảng.

virtual long getRoute(unsigned k) const;

virtual void setRoute(unsigned k, long route);

Mảng động

message FooPacket

{

fields:

long route[];

};

Trong trường hợp này, lớp C++ được sinh ra sẽ có thêm hai hàm, ngoài các hàm setter và getter bình thường. Một hàm để đặt kích thước của mảng và hàm còn lại trả về kích thước hiện tại của mảng.

virtual long getRoute(unsigned k) const;

virtual void setRoute(unsigned k, long route);

virtual unsigned getRouteArraySize() const;

virtual void setRouteArraySize(unsigned n);

Hàm set...ArraySize() cấp phát bộ nhớ cho một mảng mới. Các giá trị tồn tại trong mảng sẽ được duy trì (được sao chép sang một mảng mới).

Kích thước mặc định của mảng là 0. Điều này có nghĩa là bạn cần phải gọi hàm set...ArraySize() trước khi bạn có thể bắt đầu nhập các phần tử của mảng.

Chuỗi

message FooPacket

{

fields:

string hostName;

};

Các hàm getter và setter sẽ có dạng như sau:

virtual const char *getHostName() const;

virtual void setHostName(const char *hostName);

Chú ý: một chuỗi khác với một mảng ký tự. Mảng ký tự được coi như là một mảng thông thường.

Ví dụ:

message FooPacket

{

fields:

char chars[10];

};

Các hàm getter và setter tương ứng sẽ là:

virtual char getChars(unsigned k);

virtual void setChars(unsigned k, char a);

6.2.4. Quan hệ kế thừa và hợp thành

Những phần trên nói về việc thêm các trường dữ liệu cơ bản (int, double, char, ...) vào một message. Đối với những module đơn giản như vậy là khá đủ tuy nhiên đối với những module phức tạp, bạn còn cần:

Thiết lập cấu trúc phân cấp cho các lớp message, nghĩa là các lớp message không chỉ kế thừa từ lớp cMessage mà còn có thể kế thừa từ những lớp do bạn tạo ra.

Các trường dữ liệu trong message không chỉ là những kiểu dữ liệu cơ bản mà nó còn có thể là các cấu trúc (struct), các lớp hoặc các kiểu dữ liệu do người dùng tự định nghĩa.

Quan hệ kế thừa giữa các lớp message

Mặc định, các lớp message đều là các lớp con kế thừa từ lớp cMessage, tuy nhiên bạn có thể sử dụng một lớp cơ sở khác thông qua từ khoá extends

message FooPacket extends FooBase

{

fields:

...

};

Theo ví dụ này, lớp C++ tương ứng sẽ có dạng như sau:

class FooPacket : public FooBase { ... };

Khai báo lớp

Cú pháp khai báo một lớp cũng tương tự như cú pháp khai báo một message chỉ khác nhau từ khóa, class thay cho message.

```
class MyClass extends cObject
{
fields:
...
};
```

Chú ý rằng nếu khai báo một lớp mà không có từ khoá extends thì lớp được tạo ra sẽ không được kế thừa từ lớp cObject. Do đó trong lớp đó sẽ không có một số hàm như name(), nameClass(), ... Để tạo một lớp có đầy đủ những hàm này nhất thiết hàm phải được khai báo extends cObject.

Khai báo cấu trúc

Bạn có thể tạo các cấu trúc "kiểu C" để sử dụng như các trường dữ liệu trong các lớp message. Cấu trúc "kiểu C" có nghĩa là chỉ chứa dữ liệu và không có hàm (trong thực tế thì cấu trúc trong C++ có thể chứa các hàm).

Cú pháp khai báo struct:

```
struct MyStruct
{
fields:
char array[10];
short version;
};
```

Cú pháp khai báo này cũng tương tự như cú pháp khai báo message. Tuy nhiên phần mã C++ sinh ra lại khác nhau. Các cấu trúc được tự động sinh ra sẽ không có các hàm setter và getter, thay vào đó các thành phần dữ liệu của struct có kiểu truy xuất là public (các dữ liệu thành phần trong message có kiểu truy xuất là private - không cho phép truy xuất từ bên ngoài). Đối với đoạn khai báo ở trên, phần mã sinh ra sẽ có dạng như sau:

```
// generated C++
struct MyStruct
{
char array[10];
short version;
};
```

Các trường của một struct có thể có kiểu dữ liệu cơ bản hoặc là một struct khác nhưng nó không thể có kiểu chuỗi hoặc chứa một lớp.

Quan hệ kế thừa cũng được hỗ trợ đối với các struct:

```
struct Base
```

{

...

};

struct MyStruct extends Base

{

...

};

Bởi vì một cấu trúc không chứa các hàm thành phần, do đó nó có một số giới hạn:

Không hỗ trợ các mảng động (không thể khai báo các hàm cấp phát bộ nhớ cho mảng).

Các trường trừu tượng ("generation gap") không được sử dụng, bởi vì chúng được xây dựng dựa trên các hàm ảo. Khái niệm trường trừu tượng (abstract field) sẽ được mô tả ở phần sau.

Sử dụng lớp và cấu trúc trong message

Nếu bạn có một cấu trúc đã khai báo có tên là IPAddress, bạn có thể sử dụng nó trong message như sau:

message FooPacket

{

fields:

IPAddress src;

};

Cấu trúc IPAddress phải được khai báo trước trong file .msg hoặc nó phải là một kiểu C++ (xem phần Announcing C++ types).

Các hàm getter và setter tương ứng:

virtual const IPAddress& getSrc() const;

virtual void setSrc(const IPAddress& src);

6.2.5. Sử dụng các kiểu có sẵn của C++

Nếu bạn muốn sử dụng các kiểu dữ liệu tự mình định nghĩa trong các message, bạn cần phải thông báo kiểu dữ liệu đó với trình biên dịch message.

Giả sử bạn có khai báo một cấu trúc có tên IPAddress trong file ipaddress.h:

// ipaddress.h

struct IPAddress

{

int byte0, byte1, byte2, byte3;

};

Để có thể sử dụng IPAddress trong message, file message (có tên là foopacket.msg) nên chứa đoạn mã sau:

```
cplusplus
{{
#include "ipaddress.h"
}};
struct IPAddress;
```

Tác dụng của ba dòng đầu tiên chỉ đơn giản là sao chép câu lệnh #include “ipaddress.h” vào trong file foopacket_m.h để trình biên dịch biết về lớp IPAddress. Trình biên dịch sẽ không cố gắng kiểm tra ý nghĩa của các đoạn text nằm trong thân của khai báo cplusplus{{ ... }}. Dòng tiếp theo sẽ chỉ rõ cho trình biên dịch IPAddress là một cấu trúc. Những thông tin này sẽ ảnh hưởng đến phần mã được sinh ra.

Tương tự như vậy trong trường hợp bạn muốn sử dụng một lớp trong message, giả sử tên lớp là sSubQueue thì dòng cuối cùng trong đoạn khai báo bạn đổi lại là:

```
class cSubQueue;
```

Cú pháp trên được sử dụng trong trường hợp các lớp đều là lớp con trực tiếp hoặc gián tiếp của lớp cObject. Nếu một lớp không có quan hệ thừa kế với lớp cObject thì bạn phải sử dụng thêm từ khoá noncobject (nếu không trình biên dịch message sẽ nhầm và file được tạo ra sẽ gây ra lỗi khi biên dịch bằng trình biên dịch của C++):

```
class noncobject IPAddress;
```

6.2.6. Thay đổi các file C++

Mẫu Generation Gap

Đôi khi bạn cần các đoạn mã tự sinh ra có thể thực hiện được nhiều hơn hoặc khác đi so với những gì mà trình biên dịch tạo thành. Lấy ví dụ, khi đặt một trường số nguyên có tên là payloadLength, có thể bạn sẽ cần chỉnh sửa chiều dài của gói tin. Tuy nhiên đoạn mã tự sinh chỉ chứa hàm setPayloadLength(), điều này không thích hợp để đáp ứng yêu cầu đặt ra.

```
void FooPacket::setPayloadLength(int payloadLength)
{
this->payloadLength = payloadLength;
}
```

Để thoả mãn yêu cầu, hàm setPayloadLength() nên có dạng như sau:

```
void FooPacket::setPayloadLength(int payloadLength)
{
int diff = payloadLength - this->payloadLength;
this->payloadLength = payloadLength;
setLength(length() + diff);
}
```

Bình thường, nhược điểm lớn nhất của việc sinh mã tự động là sự khó khăn khi thoả mã các yêu cầu của người sử dụng. Việc chỉnh sửa bằng tay lại các file tự động này là

vô ích bởi vì chúng sẽ được viết chồng lại và những thay đổi sẽ biến mất khi các file này được tự động tạo lại.

Tuy nhiên, việc lập trình hướng đối tượng có thể giải quyết được vấn đề này. Một lớp được tự động sinh ra có thể dễ dàng thay đổi thông qua các lớp con của nó. Ta có thể định nghĩa lại bất kỳ hàm nào trong lớp con cho phù hợp với mục đích của mình.

Quá trình này được gọi là thiết kế mẫu Generation Gap.

Cú pháp:

message FooPacket

{

properties:

customize = true;

fields:

int payloadLength;

};

Thuộc tính customize cho phép sử dụng mẫu Generation Gap.

Với đoạn khai báo trên, trình biên dịch message sẽ tạo ra lớp FooPacket_Base chứ không phải là lớp FooPacket như bình thường. Khi đó để thay đổi các hàm bạn sẽ phải tạo một lớp con từ lớp FooPacket_Base, ta gọi là lớp FooPacket.

class FooPacket_Base : public cMessage

{ protecte

d: int src;

// make constructors protected to avoid instantiation

FooPacket_Base(const char *name=NULL);

FooPacket_Base(const FooPacket_Base& other);

public:

...

virtual int getSrc() const;

virtual void setSrc(int src);

};

Tuy vậy cũng không có nhiều hàm có thể được viết lại trong lớp FooPacket (có nhiều hàm không cho phép viết lại như các hàm khởi tạo do tính chất của quan hệ kế thừa):

class FooPacket : public FooPacket_Base

{

public:

FooPacket(const char *name=NULL): FooPacket_Base(name){}

FooPacket(const FooPacket& other): FooPacket_Base(other){}

```
FooPacket& operator=(const FooPacket& other)
{ FooPacket_Base::operator=(other)
; return *this;
}
virtual cObject *dup()
{
return new FooPacket(*this);
}
};
Register_Class(FooPacket);
```

Quay trở về với ví dụ về thay đổi chiều dài gói tin, ta có thể viết đoạn mã như sau:

```
class FooPacket : public FooPacket_Base
{
// here come the mandatory methods: constructor,
// copy contructor, operator=(), dup()
// ...
virtual void setPayloadLength(int newlength);
}
void FooPacket::setPayloadLength(int newlength)
{
// adjust message length
setLength(length()-getPayloadLength()+newlength);
// set the new length
FooPacket_Base::setPayloadLength(newlength);
}
```

Trường trừu tượng

Mục đích của trường trừu tượng là cho phép người sử dụng có thể viết chồng lại cách thức giá trị của trường được lưu trữ bên trong lớp. Bạn có thể khai báo bất kỳ trường nào là trường trừu tượng với cú pháp sau:

```
message FooPacket
{
properties:
customize = true;
fields:
```

```
abstract bool urgentBit;
};
```

Với một trường trừu tượng, trình biên dịch sẽ tạo ra một thuộc tính của lớp không có dữ liệu thành phần và các hàm getter/setter sẽ trở thành các hàm thuần ảo (các hàm thuần ảo là các hàm không có thân hàm):

```
virtual bool getUrgentBit() const = 0;
virtual void setUrgentBit(bool urgentBit) = 0;
```

6.2.7. Sử dụng STL trong các lớp message

Việc sử dụng các trường trừu tượng còn cho phép người sử dụng có thể dùng các lớp vector hoặc stack trong lớp message.

Xét định nghĩa message dưới đây:

```
struct Item
{ field
s: int
a;
double b;
}
message STLMessage
{
properties:
customize=true;
fields:
abstract Item foo[]; // will use vector<Item>
abstract Item bar[]; // will use stack<Item>
}
```

Nếu bạn biên dịch đoạn khai báo trên, phần mã sinh ra sẽ chỉ bao gồm một cặp hàm tương ứng với hai trường foo và bar, không có thành phần dữ liệu và không có hàm nào được khai báo cụ thể. Bạn có thể thay đổi lại mọi thứ, có thể triển khai lại các thuộc tính foo và bar với kiểu std::vector và std::stack:

```
#include <vector>
#include <stack>
#include "stlmessage_m.h"

class STLMessage : public STLMessage_Base
{
```

```
protected:
std::vector<Item>  foo;
std::stack<Item>    bar;
public:
STLMessage(const char *name=NULL, int kind=0) : STLMessage_Base(name,kind)
{}
STLMessage(const STLMessage& other) : STLMessage_Base(other.name())
{operator=(other);}
STLMessage& operator=(const STLMessage& other) {
if (&other==this) return *this;
STLMessage_Base::operator=(other);
foo = other.foo;
bar = other.bar;
return *this;
}
virtual cObject *dup() {return new STLMessage(*this);}
// foo methods
virtual void setFooArraySize(unsigned int size) {}
virtual unsigned int getFooArraySize() const {return foo.size();}
virtual Item& getFoo(unsigned int k) {return foo[k];}
virtual void setFoo(unsigned int k, const Item& afoo) {foo[k]=afoo;}
virtual void addToFoo(const Item& afoo) {foo.push_back(afoo);}
// bar methods
virtual void setBarArraySize(unsigned int size) {}
virtual unsigned int getBarArraySize() const {return bar.size();}
virtual Item& getBar(unsigned int k) {throw new cRuntimeException("sorry");}
virtual void setBar(unsigned int k, const Item& bar)
{throw new cRuntimeException("sorry");}
virtual void barPush(const Item& abar) {bar.push(abar);}
virtual void barPop() {bar.pop();}
virtual Item& barTop() {return bar.top();}
};
Register_Class(STLMessage);
```

Một số chú ý:

Các hàm setFooArraySize(), setBarArraySize() là thừa.

Hàm getBar(int k) không thể được triển khai vì kiểu stack không cho phép truy nhập các phần tử theo chỉ số.

Hàm setBar(int k, const Item&) cũng không được triển khai cùng với lý do như trên.

7. CHẠY CÁC ỨNG DỤNG OMNeT++

Như đã trình bày ở phần mở đầu, một hệ thống mạng mô phỏng trong OMNeT++ gồm các thành phần sau:

- Các file .ned mô tả topo mạng.
- Các file có phần mở rộng .msg chứa khai báo các message
- Các file C++ (có phần mở rộng là .cc trong UNIX hoặc .cpp trong Windows)

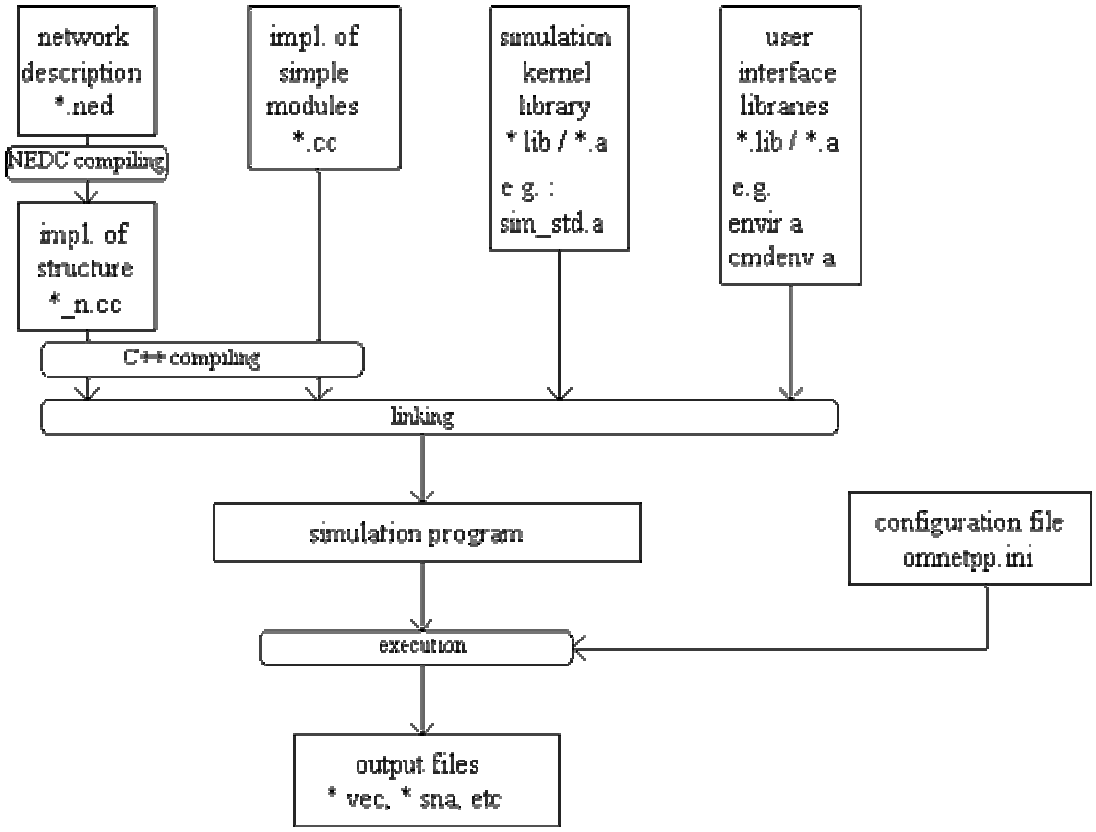
Quá trình xây dựng một chương trình mô phỏng:

- Đầu tiên, dịch các file NED và các file message thành C++, sử dụng NED compiler (nedc) và message compiler (opp_msgc).
- Quá trình tiếp theo giống như biên dịch mã nguồn C/C++:
  - Trong Linux: các file .cc ৻ file .o.
  - Trong Windows: các file .cpp ৻ file .obj.
  - Sau đó, tất cả các file trên sẽ được liên kết (link) với các thư viện cần thiết để tạo thành file .exe .

Cụ thể, ta cần phải liên kết với các thư viện sau:

- Phần nhân mô phỏng được gọi là sim_std (như các file libsim_std.a, sim_std.lib, etc).
- Giao diện người dùng: cung cấp thư viện môi trường (file libenvir.a, etc) và các tiện ích tkenv và cmdenv (libtkenv.a, libcmdenv.a, etc). Các file .o (hoặc .obj) phải được liên kết tới thư viện môi trường cùng với hoặc tkenv hoặc cmdenv.

Hình dưới đây cho chúng ta hình ảnh quá trình xử lý khi mô hình được xây dựng và hoạt động.



Hình I-7.1 - Quá trình xây dựng và thực hiện mô hình

7.1 Sử dụng gcc

Tạo Makefile:

Sau khi xây dựng xong các file nguồn *.ned, *.msg, *.cc, *.h trong cùng 1 thư mục hãy gõ

```
% opp_makemake
```

Lệnh này sẽ tạo ra file có tên là **Makefile**. Sau đó gõ tiếp **make** để tạo file chạy. Tên của file chạy này sẽ trùng với tên thư mục chứa các file nguồn.

7.2 Sử dụng Microsoft Visual C++

Tương tự như trong UNIX. Nếu gõ **opp_nmakemake** trong thư mục chứa các file nguồn *.ned, *.msg, *.cpp, *.h thì sẽ tạo ra file **Makefile.vc**.

```
opp_nmakemake
```

Để tạo file chạy, ta gõ

```
nmake -f Makefile.vc
```

8. MÔ HÌNH ĐƠN GIẢN - TICTOC

Để minh họa cách mô phỏng mạng bằng OMNeT++, ta sẽ bắt đầu với 1 mạng rất đơn giản, chỉ gồm 2 nút trao đổi dữ liệu: 1 nút tạo ra gói tin và nút thứ 2 nhận và gửi trả lại gói tin này. Chúng ta gọi 2 nút này là "tic" và "toc".

Các bước để xây dựng ứng dụng trên:

1. Tạo 1 thư mục tên là tictoc và cd tới thư mục này.

2. Tạo file topo mạng. đặt tên là tictoc1.ned:

```
simple Txc1
    gates:
        in: in;
        out: out;
endsimple

//
// Two instances (tic and toc) of Txc1 connected both ways.
// Tic and toc will pass messages to one another.
//
module Tictoc1
    submodules:
        tic: Txc1;
        toc: Txc1;
    connections:
        tic.out --> delay 100ms --> toc.in;
        tic.in <-- delay 100ms <-- toc.out;
endmodule

network tictoc1 : Tictoc1
endnetwork
```

Mô tả:

- Ta đã định nghĩa 1 mạng gọi là tictoc1là một đối tượng của module kiểu Tictoc1

  (network

  ….

  endnetwork).

- Tictoc1 là 1 module tổng hợp, gồm 2 submodules, tic và toc. 2 module thành phần này đều là các đối tượng của module có kiểu **Txc1**. Sau đó, ta kết nối cổng ra của tic tới cổng vào của toc và ngược lại. Độ trễ khi truyền là 100ms.
- **Txc1** là 1 module đơn (tức là nó có cấp độ thấp nhất trong NED file và được viết bằng C++). **Txc1** có 1 cổng vào tên là "in" và 1 cổng ra đặt tên là "out"

  (simple

  …..

  endsimple).

3. C++ file **txc1.cc** (UNIX) hoặc txc1.cpp trong Windows

```
#include <string.h>
#include <omnetpp.h>
class Txc1 : public cSimpleModule
{
  // This is a macro; it expands to constructor definition.
  Module_Class_Members(Txc1, cSimpleModule, 0);
  // The following redefined virtual function holds the algorithm.
  virtual void initialize();
  virtual void handleMessage(cMessage *msg);
};

// The module class needs to be registered with OMNeT++
Define_Module(Txc1);

void Txc1::initialize()
{
  // Initialize is called at the beginning of the simulation.
  // To bootstrap the tic-toc-tic-toc process, one of the modules needs
  // to send the first message. Let this be `tic'.

  // Am I Tic or Toc?
  if (strcmp("tic", name()) == 0)
  {
    // create and send first message on gate "out". "tictocMsg" is an
    // arbitrary string which will be the name of the message object.
    cMessage *msg = new cMessage("tictocMsg");
    send(msg, "out");
  }
```

```
}

void Txc1::handleMessage(cMessage *msg)
{
    // The handleMessage() method is called whenever a message arrives
    // at the module. Here, we just send it to the other module, through
    // gate `out'. Because both `tic' and `toc' does the same, the message
    // will bounce between the two.
    send(msg, "out");
}
```

Lớp **Txc1** kế thừa từ **cSimpleModule** và 2 phương thức initialize(), handleMessage(). Các chỉ dẫn cụ thể đã được trình bày ở phần [5.5] và [5.6]

4. Tạo Makefile:

- Trong UNIX:

```
$ opp_makemake
```

- Trong Windows (VC++)

```
D:\tictoc>opp_nmakemake
Creating Makefile.vc in D:/tictoc...
Makefile.vc created.
Please type `nmake -f Makefile.vc depend' NOW to add dependencies!
```

5. Biên dịch và liên kết để tạo file chạy

- UNIX

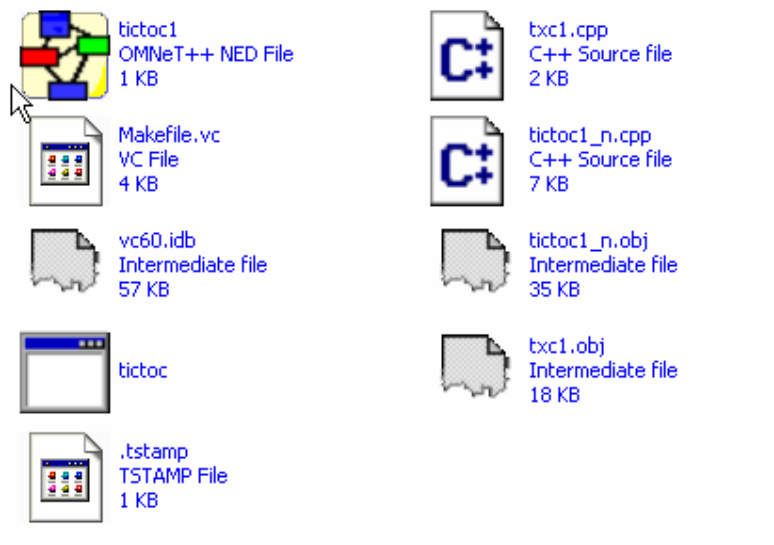
```
$ make
```

- Windows+MSVC:

```
>nmake –f Makefile.vc
```

```
D:\tictoc>nmake -f Makefile.vc

Microsoft (R) Program Maintenance Utility   Version 6.00.8168.0
Copyright (C) Microsoft Corp 1988-1998. All rights reserved.

        C:\OMNeT++\bin\nedtool -s _n.cpp  tictoc1.ned
        cl.exe -c /nologo /GX /GR /FD /Zm200 /ML /O2 /D "NDEBUG"   -IC:/OMNeT++/
include /Tp tictoc1_n.cpp
tictoc1_n.cpp
        cl.exe -c /nologo /GX /GR /FD /Zm200 /ML /O2 /D "NDEBUG"  -IC:/OMNeT++/i
nclude /Tp txc1.cpp
txc1.cpp
        link.exe /nologo /subsystem:console /machine:I386 /pdbtype:sept /increme
ntal:no tictoc1_n.obj txc1.obj    /libpath:C:/OMNeT++/lib envir.lib tkenv.lib tc
l84.lib tk84.lib /libpath:"C:/OMNeT++/lib" sim_std.lib nedxml.lib libxml2.lib ic
onv.lib wsock32.lib   /out:tictoc.exe
```

Ta sẽ thấy có thêm các file trong thư mục tictoc



6. Muốn chạy được, ta còn phải tạo file **omnetpp.ini** để báo cho trình mô phỏng biết ta muốn chạy mô phỏng đối với mạng nào. Điều này cũng có nghĩa nhiều mạng có thể chung 1 trình mô phỏng.

File omnetpp.ini trong ví dụ này cũng rất đơn giản

```
[General]
network = tictoc1
```

7. Sau đó, ta sẽ bắt đầu chạy ứng dụng bằng lệnh:

-UNIX

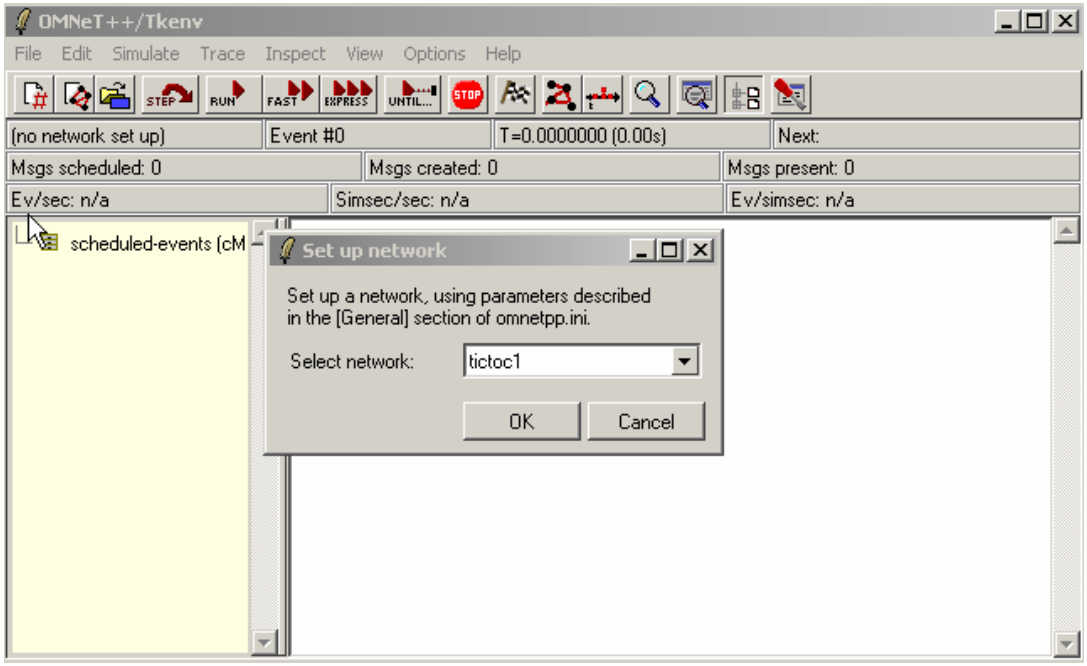
```
$ ./tictoc
```

- Windows

```
>tictoc
```

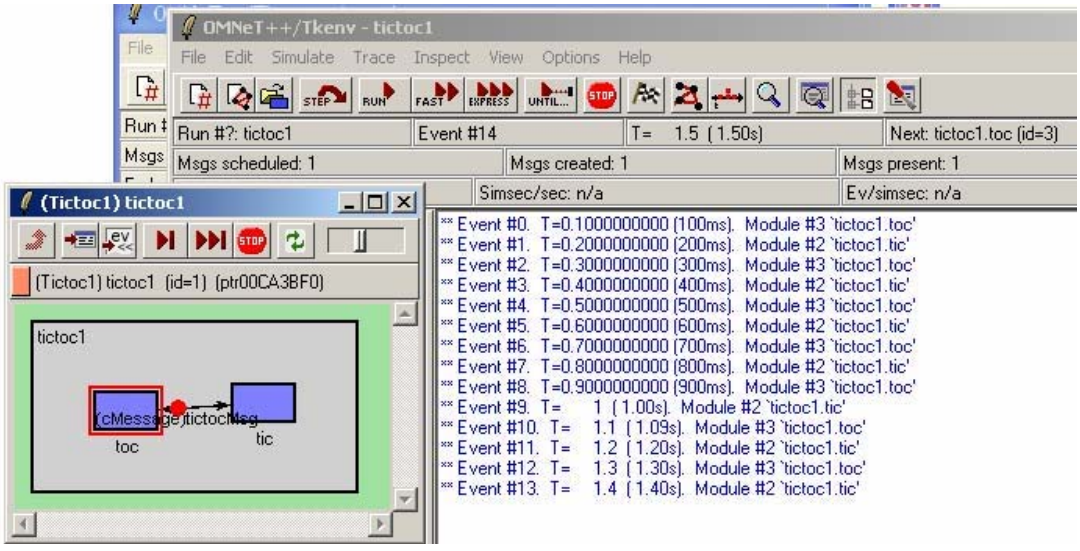
```
D:\tictoc>tictoc
OMNeT++/OMNEST Discrete Event Simulation  (C) 1992-2004 Andras Varga
See the license for distribution terms and warranty disclaimer
Setting up Tkenv (Tk-based graphical user interface)...

Loading bitmaps from .\bitmaps: *: 0
Loading bitmaps from C:\OMNeT++\bitmaps: *: 0  abstract/*: 72  block/*: 256  dev
ice/*: 156  msg/*: 20  old/*: 111  place/*: 56  status/*: 17

Plugin path: ./plugins
```

Ta sẽ thấy cửa sổ mô phỏng hiện ra



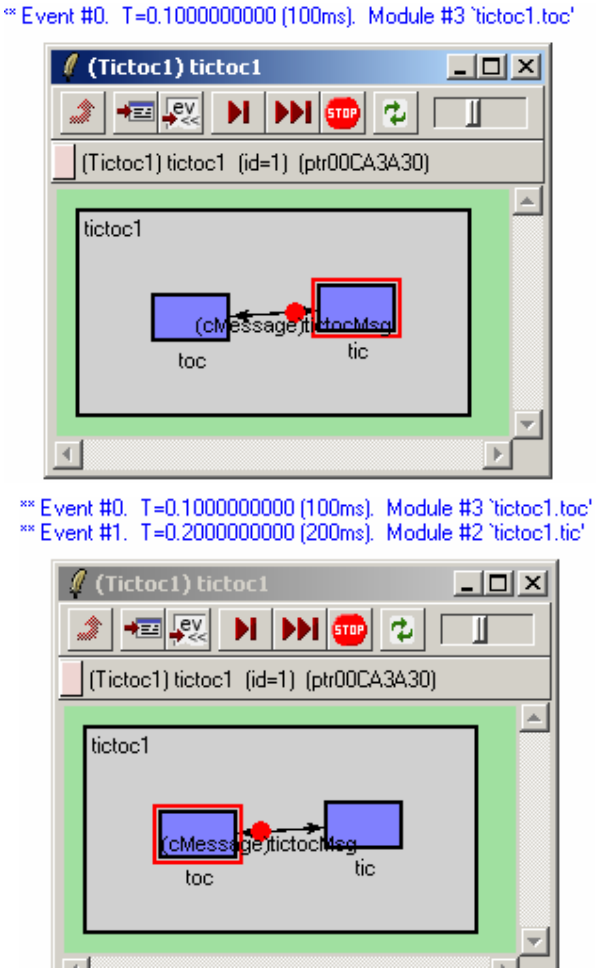
Click OK

8. Nhấn nút Run bắt đầu quá trình mô phỏng để thấy 2 nút mạng trao đổi các gói tin cho nhau.



Cửa sổ chính hiển thị thời gian mô phỏng. Nó hiện thời điểm ứng với mỗi lần truyền tin.

9. Ta có thể hiệu chỉnh chế độ chạy: dừng - F8 (STOP button), chạy từng bước F4) … Ví dụ: Step mode



10. Muốn thoát chỉ cần ấn Close hoặc chọn File|Exit.