PHẦN I - TỔNG QUAN VỀ OMNET++ 5
1. GIỚI THIỆU 5
1.1. OMNeT++ là gì? 5
1.2. Các thành phần chính của OMNeT++ 5
1.3. Ứng dụng 5
1.4. Mô hình trong OMNeT++ 6
2. TỔNG QUAN 7
2.1. Khái niệm mô hình hoá 7
2.1.1. Cấu trúc phân cấp của các module 7
2.1.2. Kiểu module 7
2.1.3. Message, cổng, liên kết 8
2.1.4. Mô hình truyền gói tin 9
2.1.5. Tham số 10
2.1.6. Phương pháp mô tả topology 10
2.2. Lập trình thuật toán 10
2.3. Sử dụng OMNeT++ 11
2.3.1. Xây dựng và chạy thử các mô hình mô phỏng 11
2.3.2. Hệ thống file 12
3. NGÔN NGỮ NED 14
3.1 Tổng quan về NED 14
3.1.1. Các thành phần của ngôn ngữ mô tả NED 14
3.1.2. Các từ khoá 14
3.1.3. Đặt tên 14
3.1.4. Chú thích 15
3.2. Các chỉ dẫn import 15
3.3. Khai báo các kênh 15
3.4. Khai báo các module đơn giản 16
3.4.1. Các tham số của module đơn giản 16
3.4.2. Các cổng của module đơn giản 17
3.5. Khai báo module kết hợp 18
3.5.1. Các tham số và cổng của module kết hợp 19
3.5.2. Các module con 19
3.5.3. Tham số tên kiểu module con 20
3.5.4. Gán giá trị cho các tham số của các module con 22
3.5.5. Khai báo kích thước của các vector cổng của module con 23
3.5.6. Khai báo gatesizes và tham số có điều kiện 23
3.5.7. Kết nối 24
3.6. Khai báo mạng 27
3.7. Các biểu thức 27
3.7.1. Hằng số 28
3.7.2. Tham chiếu 28
3.7.3. Các toán tử 29
3.7.4. Toán tử sizeof() và index 30
3.7.5. Toán tử xmldoc() 30
3.7.6. XML và XPath 31
3.7.7. Hàm 31
3.7.8. Giá trị ngẫu nhiên 32
3.7.9. Khai báo một hàm mới 33
4. GIỚI THIỆU GNED 35
4.1. Giao diện 35

4.2. Một số thao tác cơ bản 38
4.3. Làm việc với nhiều file NED - Các chức năng chỉnh sửa nâng cao 43
5. MODULE ĐƠN GIẢN 50
5.1 Module đơn giản trong OMNeT++ 50
5.2 Các sự kiện trong OMNeT++ 50
5.3 Mô hình hoá hoạt động truyền gói tin 50
5.4 Khai báo kiểu module đơn giản 51
5.4.1 Tổng quan 51
5.4.2 Đăng ký kiểu module 52
5.5 Xây dựng hàm cho Module 52
5.5.1 Hàm handleMessage() 52
5.5.2 Hàm activity() 53
5.5.3 Hàm initialize() và finish() 54
5.6 Gửi và nhận các message 54
5.6.1 Gửi các message 54
5.6.2 Broadcasts 54
5.6.3 Gửi có độ trễ (Delayed sending) 55
5.6.4 Gửi trực tiếp message 55
5.6.5 Gửi định kỳ 55
5.7 Truy nhập các cổng và kết nối 55
5.7.1 Đối tượng cổng (gate object) 55
5.7.2 Các tham số kết nối 56
5.8 Tự động tạo module 56
6. MESSAGE 58
6.1. Message và Packet 58
6.1.1. Lớp cMessage 58
6.1.2. Self-Message 59
6.1.3. Mô hình hoá gói tin 60
6.1.4. Đóng gói (Encapsulation) 62
6.1.5. Thêm đối tượng và tham số 63
6.2. Định nghĩa message 64
6.2.1. Giới thiệu 64
6.2.2. Sử dụng enum 66
6.2.3. Khởi tạo cho một message 66
6.2.4. Quan hệ kế thừa và hợp thành 69
6.2.5. Sử dụng các kiểu có sẵn của C++ 71
6.2.6. Thay đổi các file C++ 72
6.2.7. Sử dụng STL trong các lớp message 75
7. CHẠY CÁC ỨNG DỤNG OMNeT++ 78
7.1 Sử dụng gcc 79
7.2 Sử dụng Microsoft Visual C++ 79
8. MÔ HÌNH ĐƠN GIẢN - TICTOC 80
Phần II – TỔNG QUAN VỀ WLAN 86
1. GIỚI THIỆU 86
1.1 Ưu điểm của mạng LAN không dây 86
1.2 Một số ứng dụng thực tế của WLAN tại Việt Nam 88
1.2.1 Ứng dụng trong Wireless LAN Telemedicine 88
1.2.2 Hệ thống WiFi VNN 90
2. CÁC MÔ HÌNH MẠNG CƠ BẢN 91
2.1 Mô hình cơ sở (Infrastructure network) 91

2.1.1 Tập hợp dịch vụ cơ bản (BSS - Basic Service Set)....92
2.1.2 Tập hợp các dịch vụ mở rộng (ESS Extended Service Set)....92
2.2 Mô hình Adhoc độc lập (Independent network)....96
3. HOẠT ĐỘNG CỦA CÁC CHUẨN LIÊN QUAN....97
3.1 Các băng tần ISM....97
3.1.1 Băng tần ISM (ISM bands)....97
3.1.2 Băng tần UNII (UNII bands)....98
3.2 Các chuẩn 802.11 (IEEE 802.11 family)....99
3.2.1 IEEE 802.11....99
3.2.2 IEEE 802.11b....99
3.2.3 IEEE 802.11a....99
3.2.4 IEEE 802.11g....99
3.2.5 Một số chuẩn khác trong họ IEEE 802.11....100
4. MÔ HÌNH CẤU TRÚC CỦA MẠNG WLAN....103
4.1 Tầng vật lý (PHY layer)....103
4.1.1 Các kỹ thuật trải phổ....104
4.1.2 Cấu trúc khung PLCP (General PLCP Frame Format)....108
4.1.3 PLCP trong dải phổ dịch tần FHSS....109
4.1.4 PLCP cho DSSS và HR/DSSS....109
4.1.5 PLCP trong OFDM....110
4.1.6 Thủ tục truyền PLCP....110
4.1.7 Thủ tục nhận PLCP....111
4.1.8 Tầng PMD (PMD sublayer)....112
4.2 Tầng kiểm soát truy nhập đường truyền – MAC....113
4.2.1 DCF - Distributed Coordination Function....113
4.2.2 PCF – Point Coordination Function....116
4.2.3 Phân tích các hoạt động cơ bản....116
4.3 Tầng mạng và các giao thức dẫn đường trong WLAN....121
4.3.1 Các giao thức tìm đường trong mạng Ad-hoc....122
4.3.2 Các giao thức mở rộng cho MANET....123
4.3.3 Mô tả chi tiết giao thức AODV....125
PHẦN III – PHÂN TÍCH THIẾT KẾ ỨNG DỤNG MÔ PHỎNG MẠNG ADHOC....130
1. MÔ HÌNH CHUNG....130
2. CẤU TRÚC HỆ THỐNG....131
2.1 Tầng vật lý (Physical model)....131
2.2 Tầng điều khiển truy nhập (Mac Layer)....131
2.3 Tầng mạng (Routing model)....133
2.4 Mobility models....136
2.5 Tầng ứng dụng....137
2.6 Liên kết giữa các tầng....138
2.7 Thiết lập các thông số cho hệ mô phỏng....138
2.7.1 Thông số của Map và Hosts....138
2.7.2 Physical Layer....139
2.7.3 Mac Layer....139
2.7.4 Routing....139
2.7.5 Application....140
3. KẾT QUẢ THỰC HIỆN....141
3.1 Topo....141
3.2 Gửi các gói tin Hello....142

3.3 Gửi gói tin RREQ 142
Phần IV - PHỤ LỤC 143
1. SO SÁNH OMNET++ VÀ NS/2 143
2.TÀI LIỆU THAM KHẢO 145

PHẦN I - TỔNG QUAN VỀ OMNET++

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

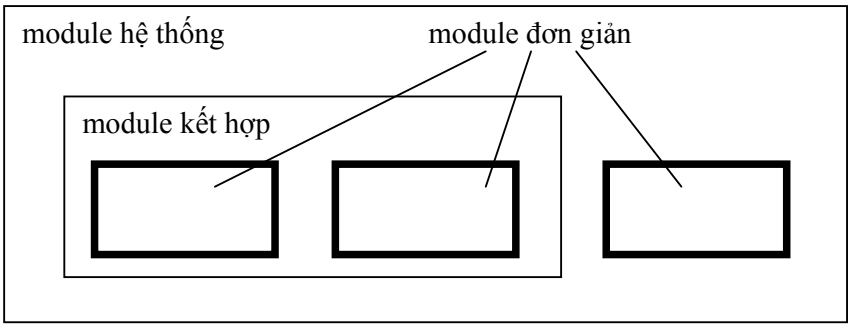
Các tham số của module linh hoạt

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

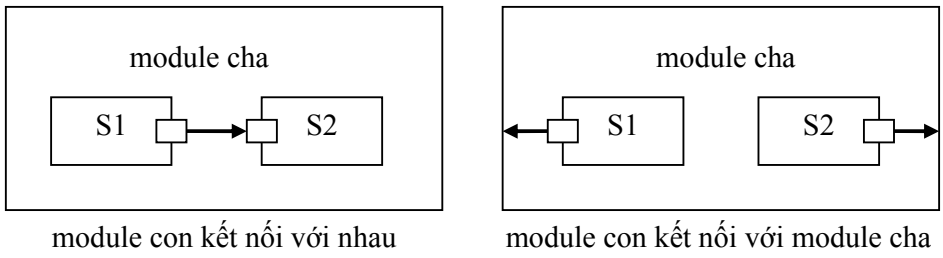
2.1.3. Message, cổng, liên kết

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.

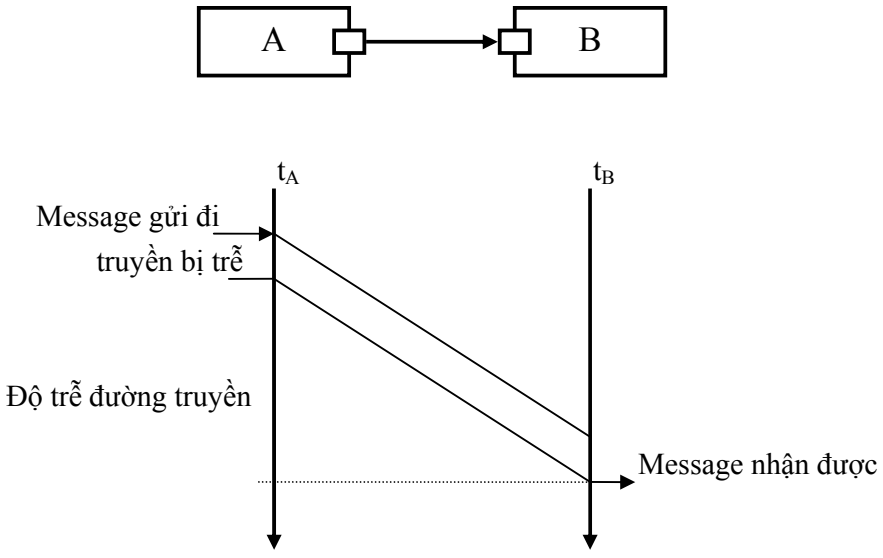
Tỉ số lỗi bit ảnh hưởng đến quá trình truyền message qua kênh. Tỉ số này là xác suất các bit bị truyền sai. Do đó xác suất để một message độ dài n bit truyền đi chính xác là:

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 thể (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

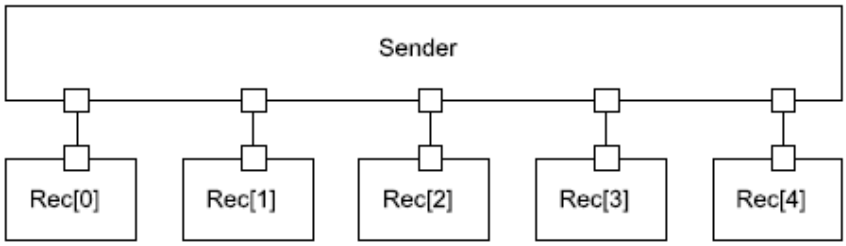
Nếu module con hoặc các vector cổng được sử dụng, NED cho phép người sử dụng có thể tạo ra nhiều hơn một kết nối với một khai báo. Đó được gọi là một đa kết nối hay một kết nối vòng lặp.

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
```

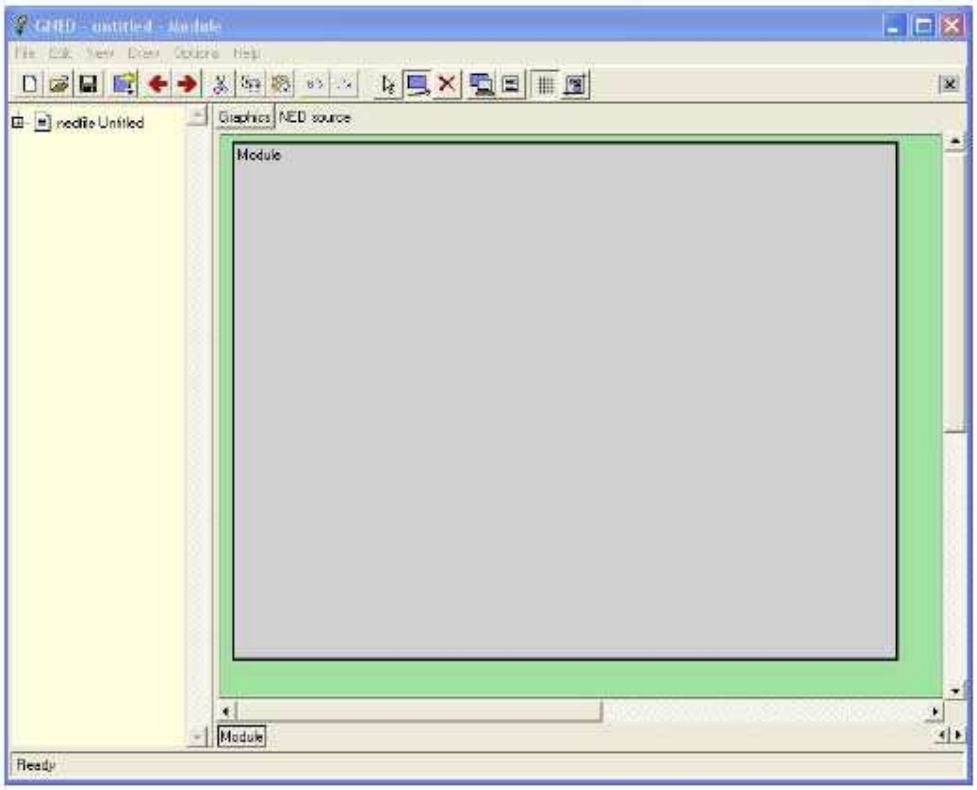
Nếu hàm nhận vào tham số có kiểu int, long hoặc một số kiểu khác không phải là kiểu double thì bạn phải tạo hàm chuyển đổi. Trong trường hợp này bạn phải đăng ký hàm chuyển đổi với macro Define_Function2() để cho phép một hàm có thể được đăng ký

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.

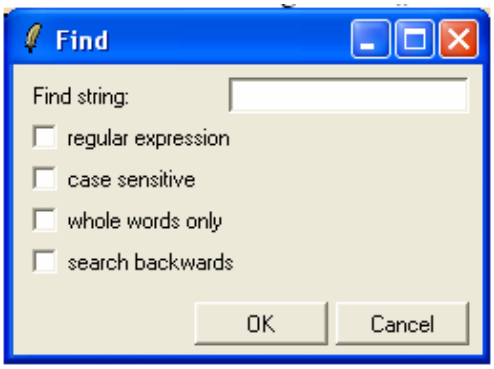
 Copy: sao chép vào clipboard.

 Paste: “dán” từ clipboard.

 Undo: huỷ bỏ thao tác vừa thực hiện.

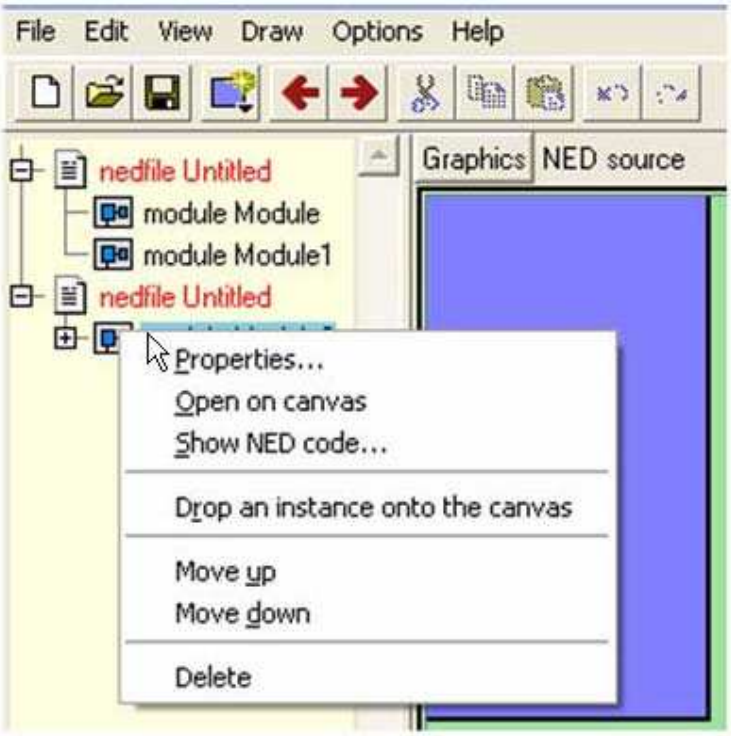
 Redo: lặp lại thao tác vừa huỷ bỏ.

 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.



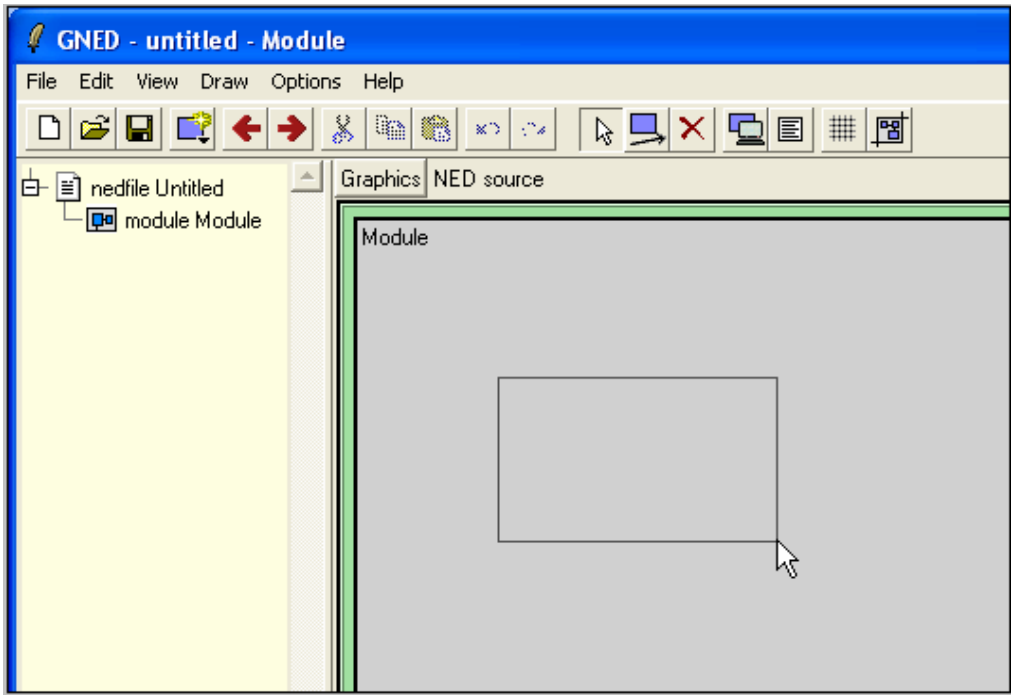
Hai mục "nedfile Untitled" được đánh dấu mầu đỏ để nhắc nhở là hai file này chưa Save.

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.

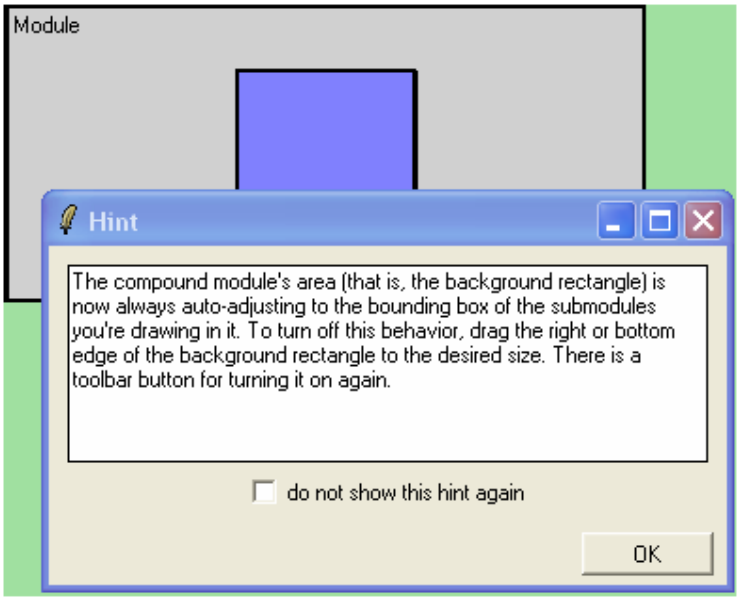
Khi bạn mới sử dụng GNED, sẽ thường có một cửa sổ nhỏ hiện lên mỗi lần bạn thực hiện một thao tác. Cửa sổ này giải thích cho bạn cách thức hoạt động của một số chức năng phức tạp trong giao diện của GNED. Bạn có tắt chức năng này bằng cách đánh dấu vào hộp chọn phía dưới của cửa sổ. Đây là một chức năng 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

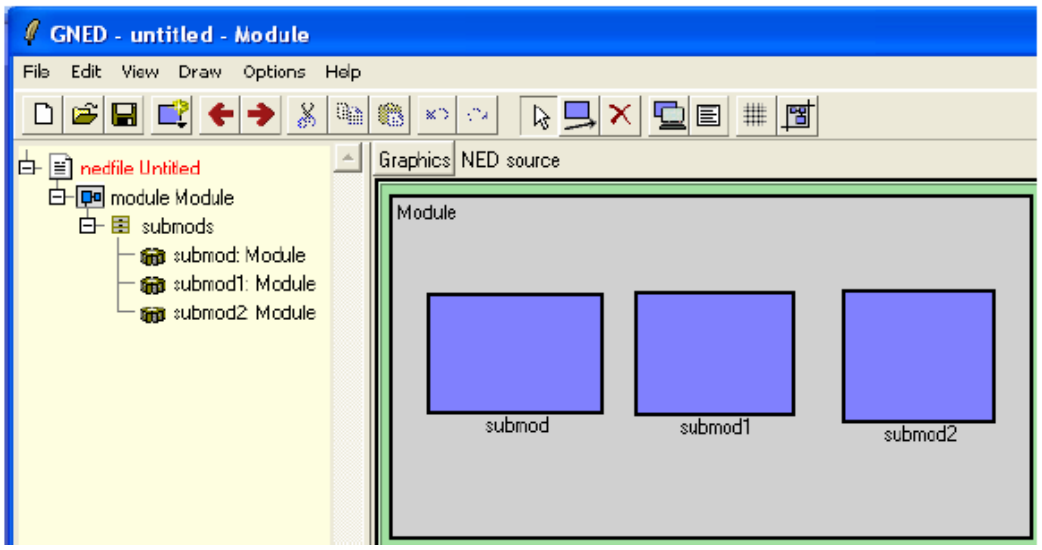
Tạo một module con trong module kết hợp



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:

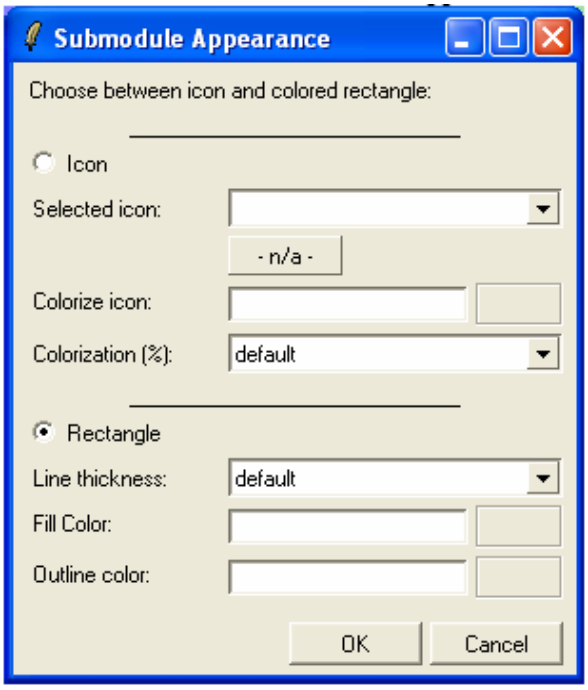


Nhấn OK, tương tự bạn tạo thêm một vài module như trong hình vẽ:

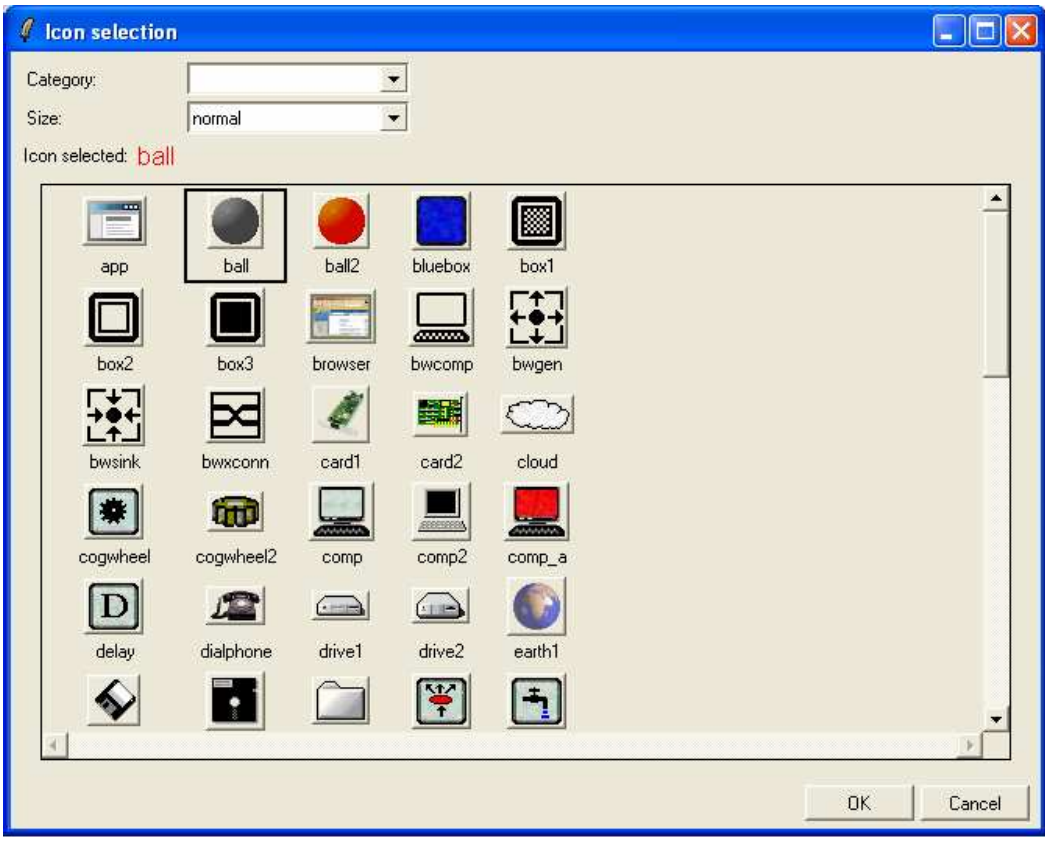


Tiếp theo ta sử dụng công cụ  để chọn một trong các module con vừa được tạo ra. Biên của module được chọn chuyển thành mầu đỏ.

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.

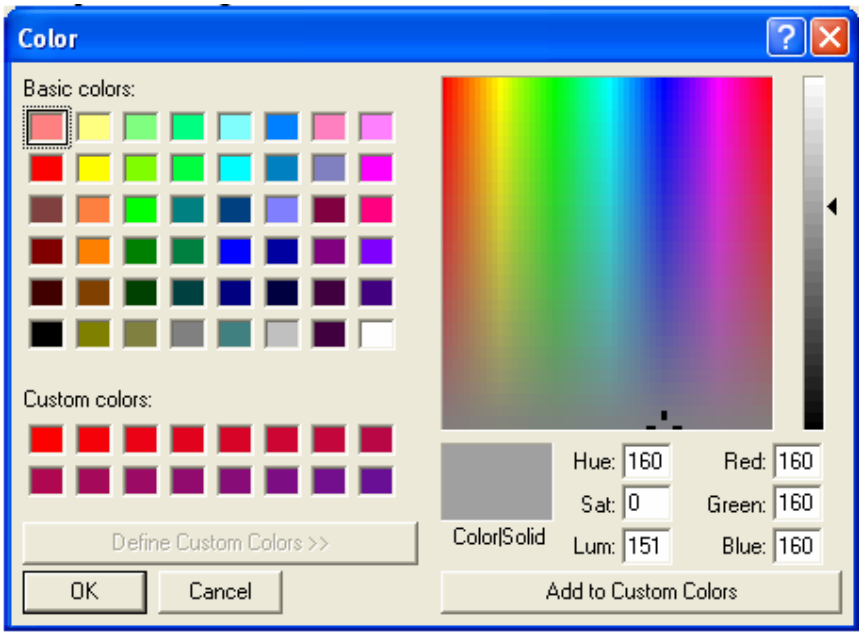


Bạn có thể sử dụng những Icon có sẵn của OMNeT++ hoặc có thể thêm các Icon của riêng mình bằng cách copy các file bitmap vào thư mục bitmap của OMNeT++.

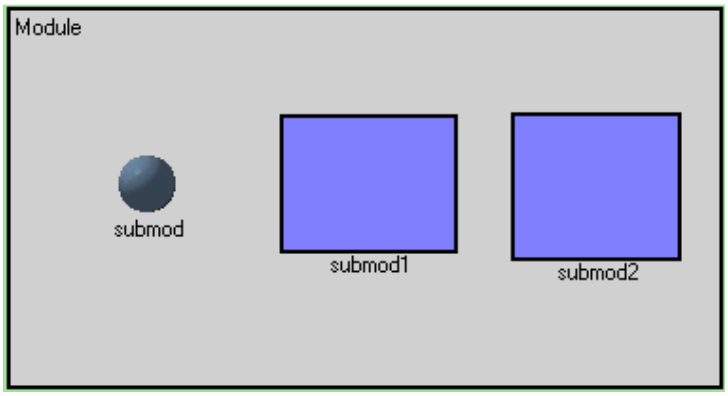
Chọn Icon "ball" với kích thước trung bình ("normal"). Ta cũng có thể thay đổi mầu sắc cho các Icon.



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



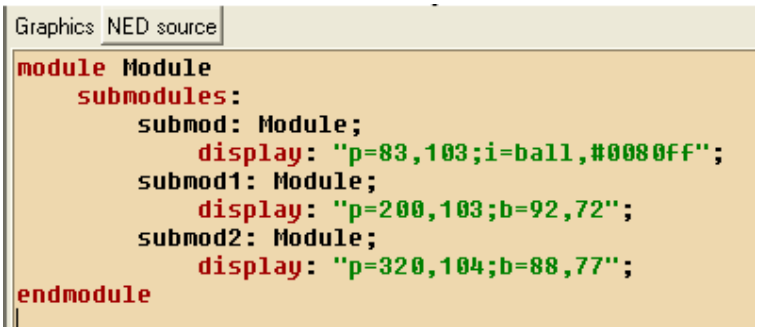
Sau khi chọn mầu xanh, bây giờ bạn có thể xem lại hình dạng của module con mà bạn vừa mới thay đổi so với hình chữ nhật của các module cũ.



Chuyển sang chế độ nhìn NED Source



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



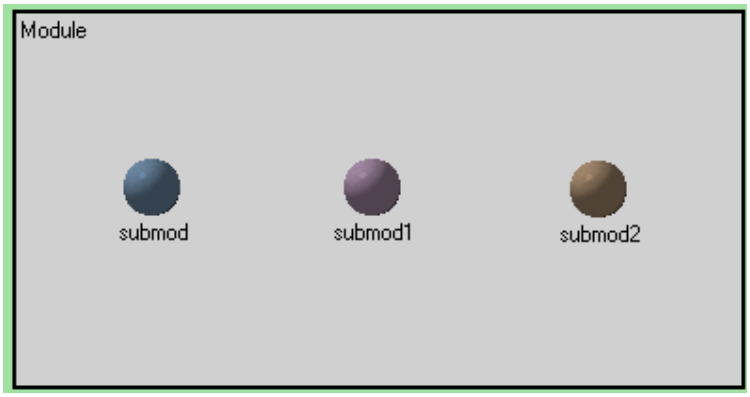
Chú ý là đối với module con thứ nhất những thay đổi về mặt hình thức của nó được diễn tả trong chuỗi "i=ball, #0080ff". Bằng cách sử dụng các chức năng copy, paste ta

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:

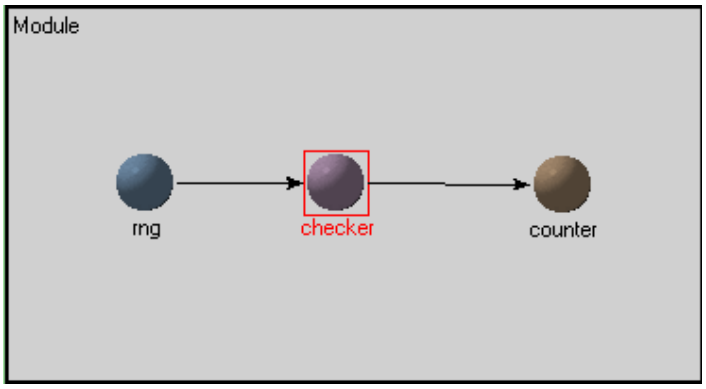


Như vậy trong GNED, hai chế độ giao diện có ảnh hưởng qua lại lẫn nhau. Những thay đổi trong chế độ này cũng ảnh hưởng đến chế độ còn lại.

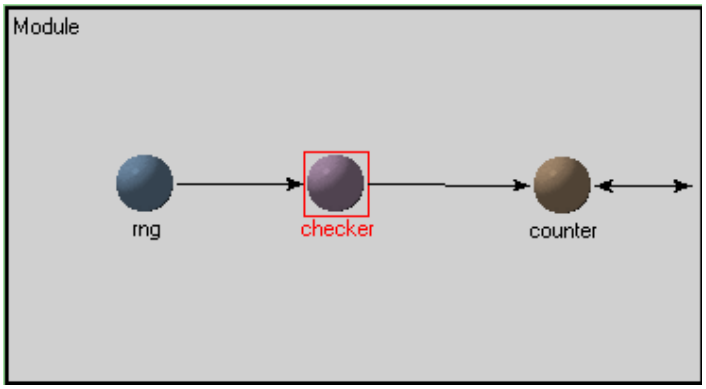
Bây giờ chúng ta sẽ sử dụng các module này để thử nghiệm chức năng của bộ tạo số ngẫu nhiên (Random Number Generator - RNG). Trước tiên, kích phải chuột vào từng module và chọn chức năng Rename để đổi tên cho các module theo thứ tự lần lượt là "rng", "checker" và "counter".

"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.

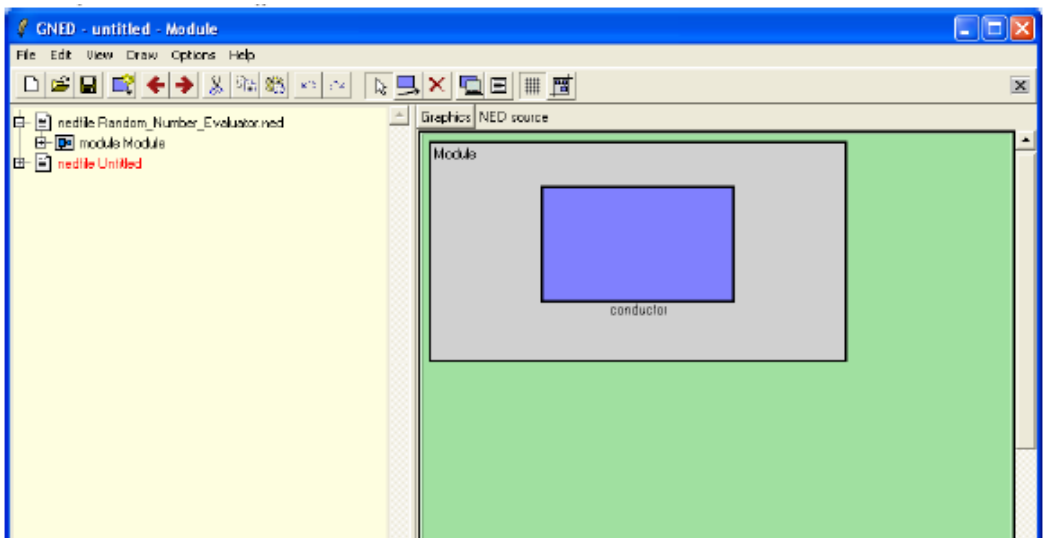


Bây giờ ta đổi tên cho module kết hợp thành "rnd_eval" bằng cách kích phải chuột vào dòng chữ "Module" nằm ở góc trên trái của vùng mầu xám, chọn chức năng Rename trong trình đơn ngữ cảnh tương ứng.

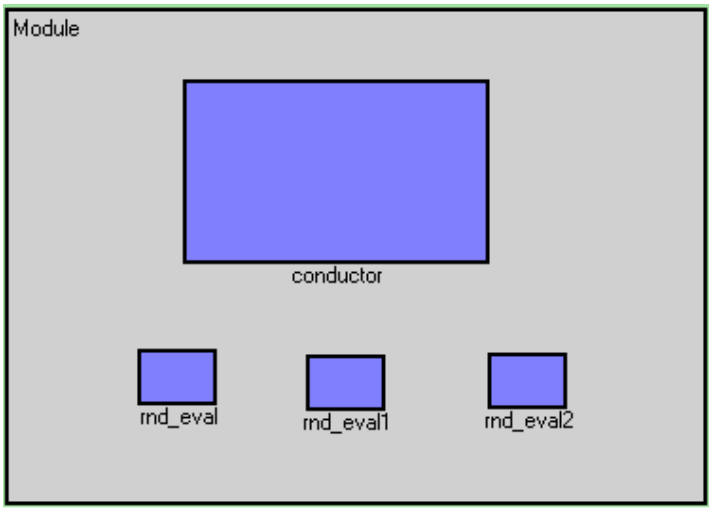
Save file với tên "Random_Number_Evaluator.ned", chúng ta đã có một file NED hoàn chỉnh.

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

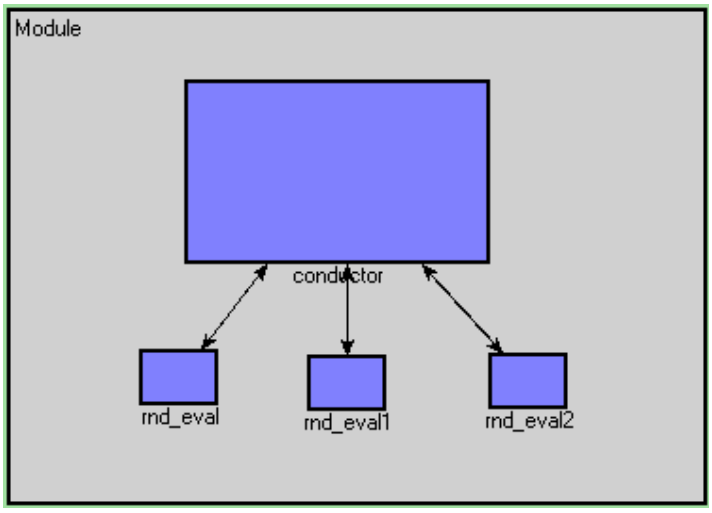
Nhấn nút  để tạo một tài liệu NED mới. Tạo một module con mới có tên là "conductor".



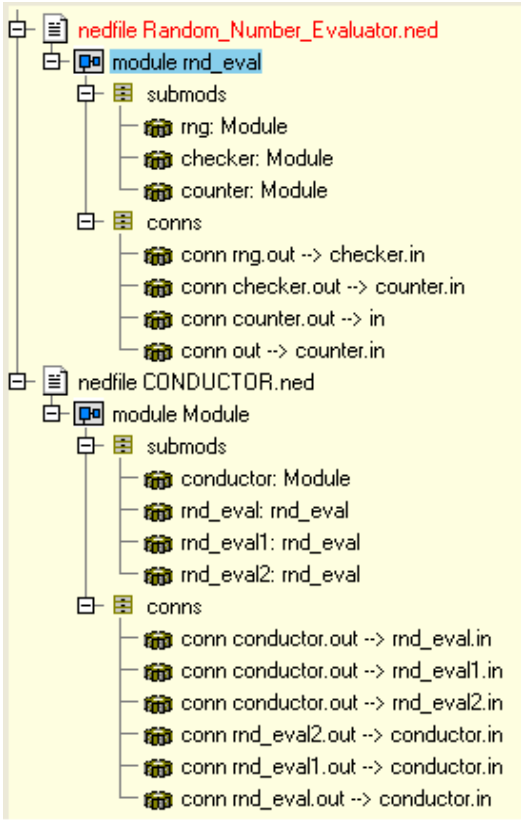
Sau đó kéo rê module "rnd_eval" của file ta vừa tạo ở trên vào module "Module" của file mới ba lần.



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



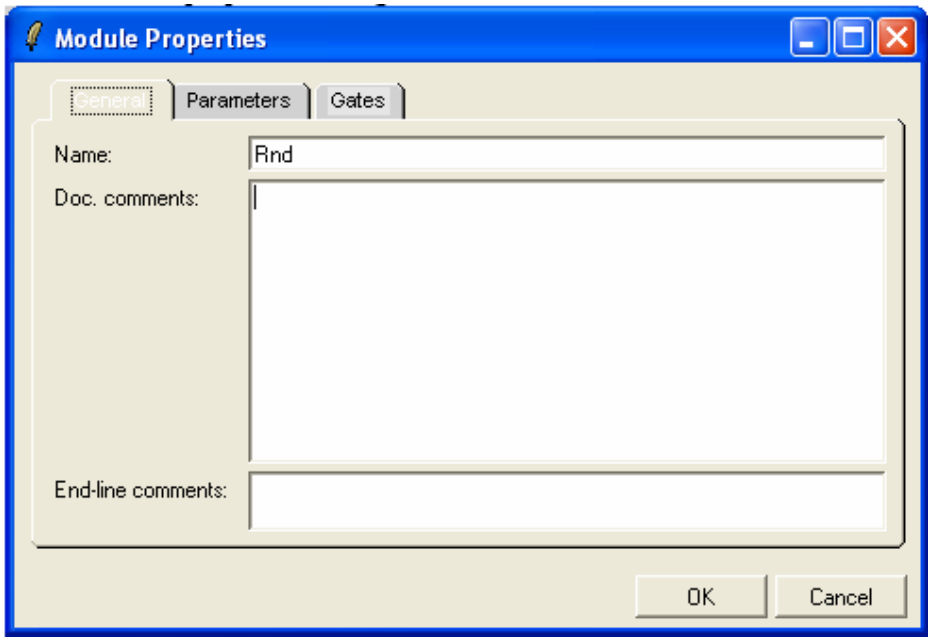
Và file NED cũng có những thay đổi tương ứng

```
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
```

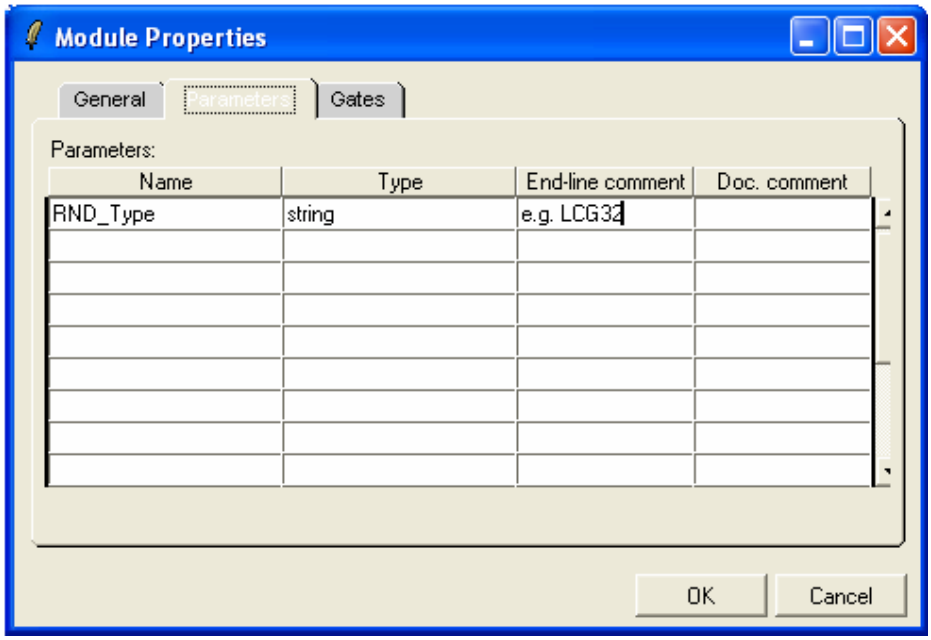
Save file mới này với tên “conductor.ned”.

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”.



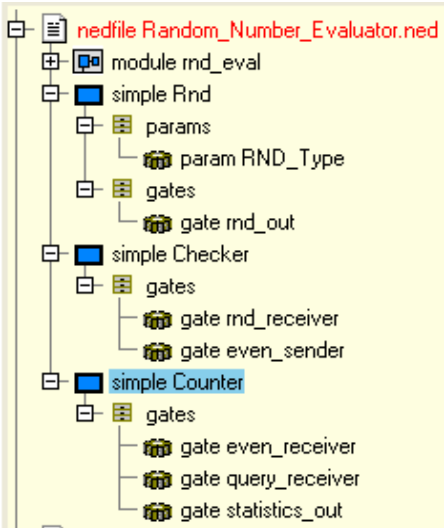
Chọn tab Parameters để định nghĩa các tham số (các tham số này có thể sẽ được các RNG sử dụng), bằng cách này bạn có thể cấu hình lại sau này.



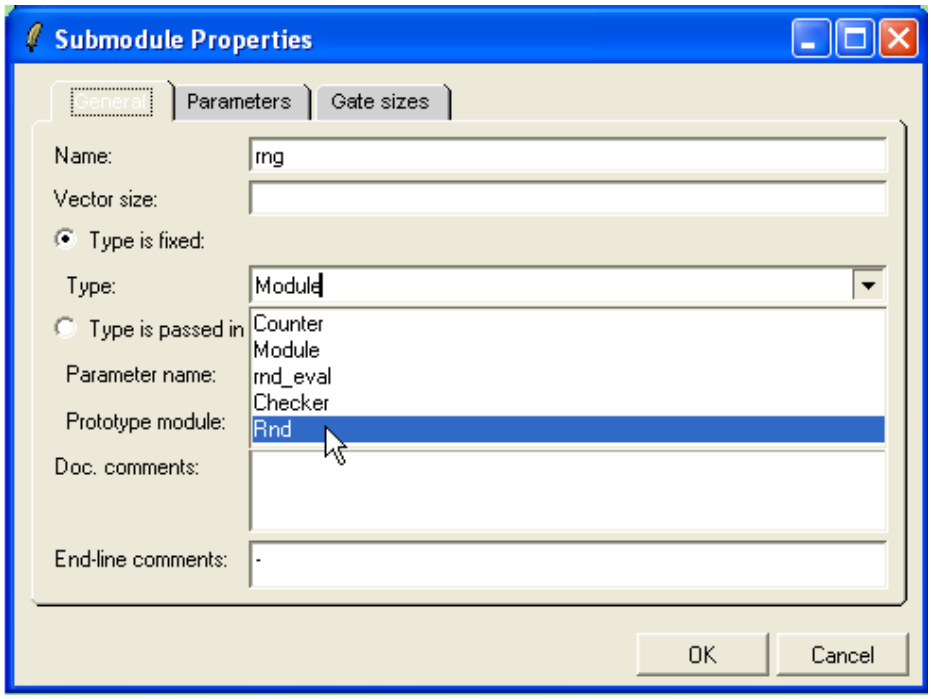
Chọn tab Gates và khai báo một cổng, qua đó các số ngẫu nhiên sinh ra sẽ được truyền đi. Bạn có thể sử dụng trường End-line comment để viết chú thích cho chức năng của cổng.

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".



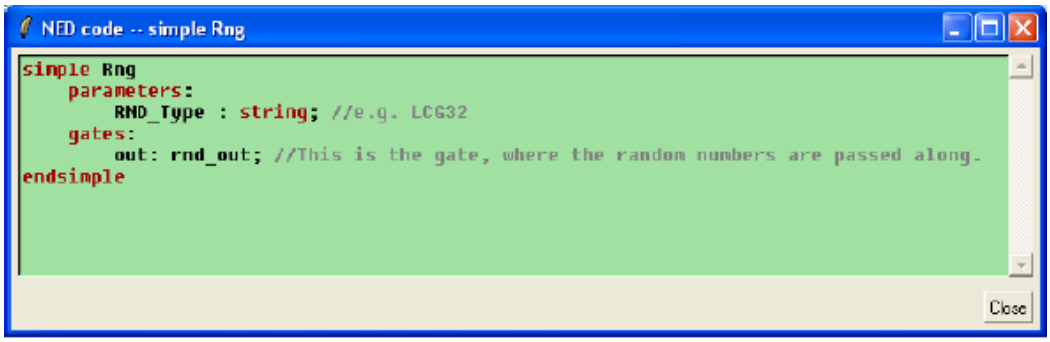
Tiếp theo ta thay đổi kiểu của mỗi module con bên trong module rnd_eval cho phù hợp với các kiểu module đơn giản mới mà ta vừa tạo ra. Kích chuột phải vào tên module "rng", chọn "Properties...", sau đó chọn trong Type kiểu Rnd.



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
```

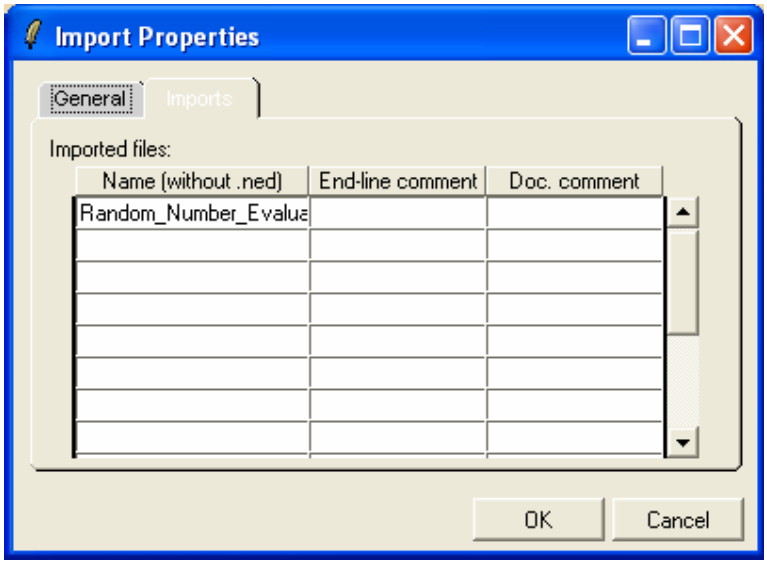
Bạn thấy rằng các module con "rng", "checker" và "counter" đã có các kiểu module tương ứng. Ngoài ra, nếu bạn kích chuột phải vào simple Rnd và chọn "Show NED code...", bạn sẽ thấy những khai báo bạn thực hiện trong cửa sổ Module Properties ở trên được thể hiện dưới dạng mã:



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
{
protected:
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
{
fields:
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)

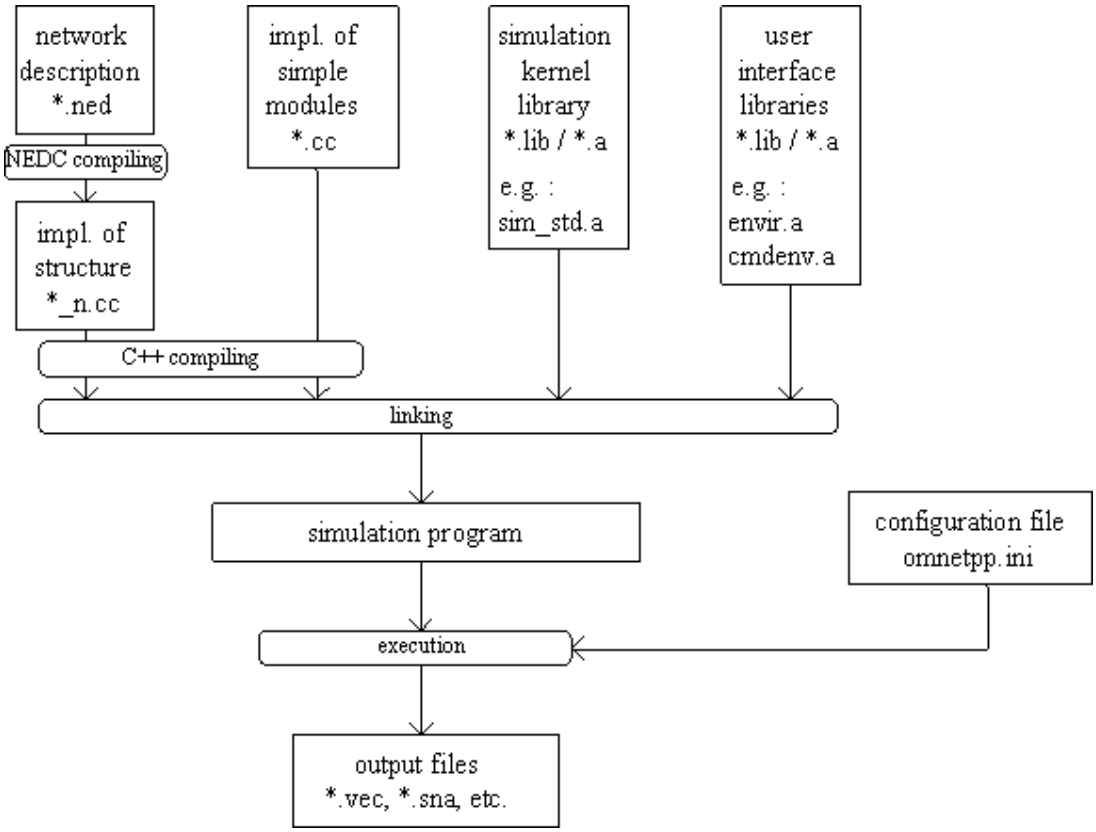
Quá trình xây dựng một chương trình mô phỏng:

- 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:

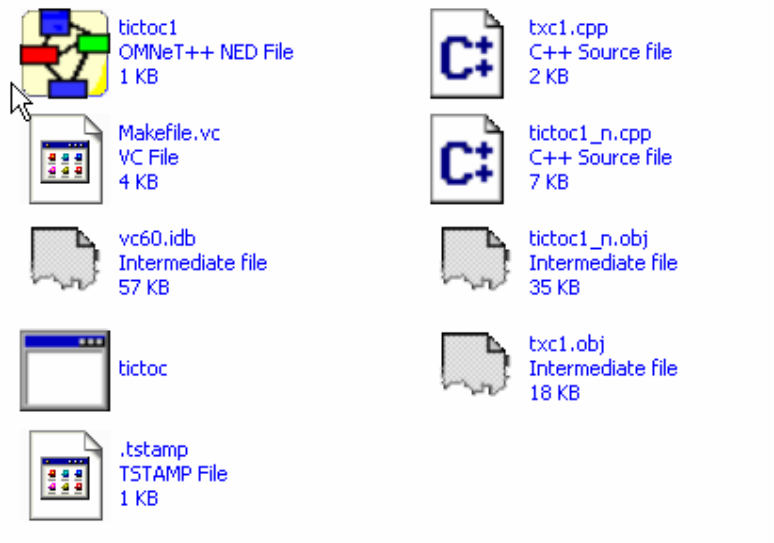
```
>nmake –f Makefile.vc
```

```
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

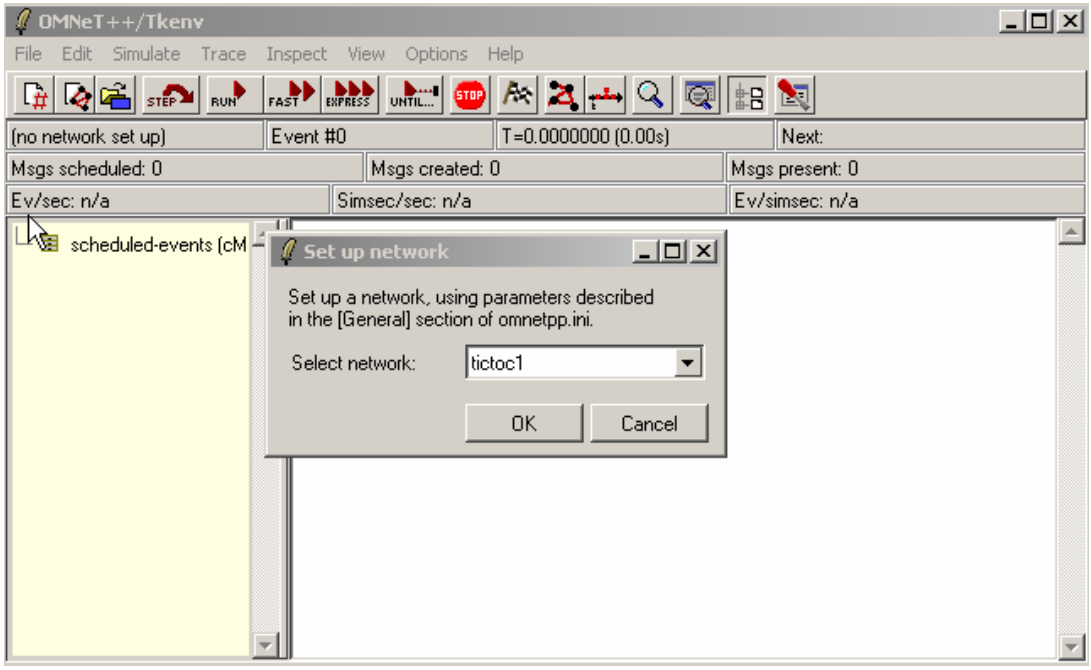
```
>tictoc
```

```
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

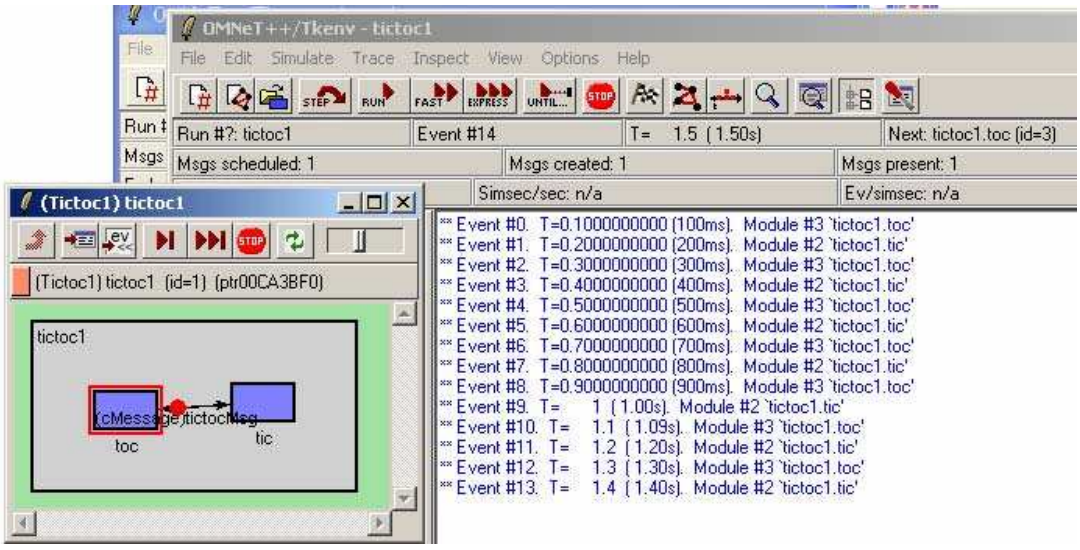
Plugin path: ./plugins
```

Ta sẽ thấy cửa sổ mô phỏng hiện ra



Click OK

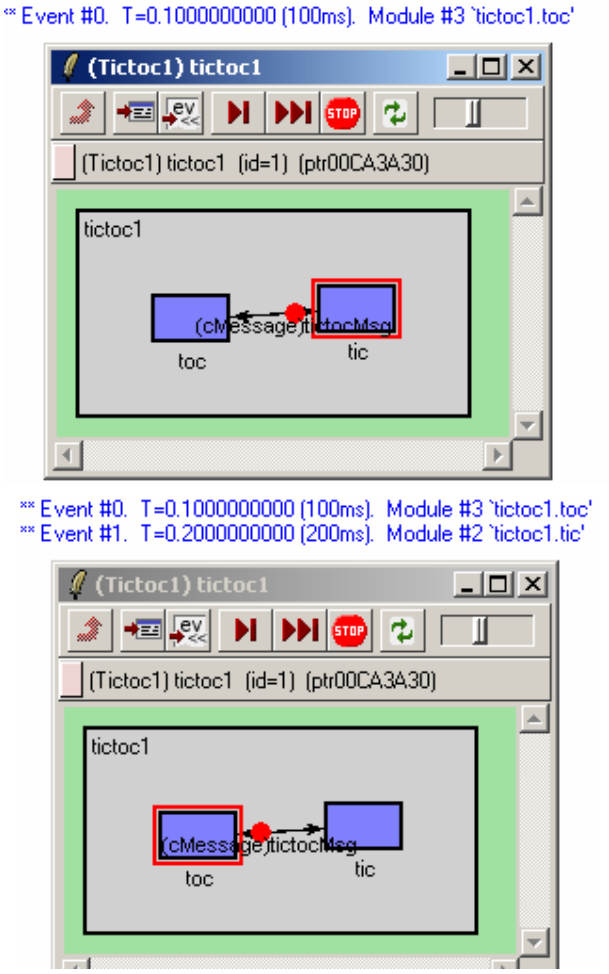
8. Nhấn nút Run bắt đầu quá trình mô phỏng để thấy 2 nút mạng trao đổi các gói tin cho nhau.



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.

Phần II – TỔNG QUAN VỀ WLAN

1. GIỚI THIỆU

Công nghệ không dây nói chung và công nghệ mạng LAN không dây (WLAN) nói riêng hiện nay đang là điểm nóng trong lĩnh vực Công nghệ thông tin và Truyền thông (ICT) toàn thế giới.

Mạng LAN không dây hay còn gọi là mạng LAN vô tuyến (Wireless LAN) là một mạng dữ liệu mềm dẻo, sử dụng sóng vô tuyến hay hồng ngoại để truyền và nhận dữ liệu. WLAN cung cấp tất cả các chức năng và ưu điểm của một mạng LAN truyền thống như Ethernet hay Ring và không bị giới hạn bởi cáp.

1.1 Ưu điểm của mạng LAN không dây

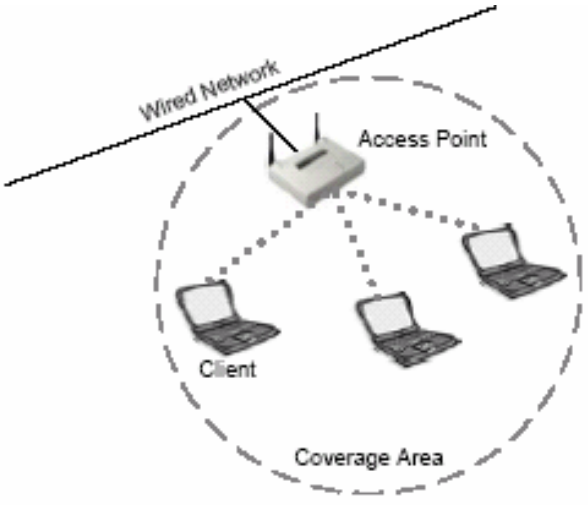
Khả năng ứng dụng rộng rãi trên phạm vi rộng, sự phát triển nhanh chóng của Internet và dịch vụ trực tuyến là lợi ích của việc chia sẻ dữ liệu và tài nguyên. Với WLAN, người dùng truy nhập vào thông tin chia sẻ không cần hệ thống dây để kết nối, không cần lắp đặt hoặc di chuyển dây khi người quản trị mạng thiết lập mở rộng mạng. So với mạng LAN truyền thống, WLAN có các ưu điểm nổi trội về hiệu suất, sự tiện lợi và chi phí xây dựng.

- Khả năng di động: hệ thống WLAN có thể cung cấp người dùng LAN truy nhập thông tin thời gian thực mọi nơi trong các tổ chức của họ. Sự linh hoạt này hỗ trợ hiệu suất và cơ hội dịch vụ không thể có ở mạng có dây.
- Dễ dàng triển khai: thiết lập hệ thống WLAN nhanh và dễ dàng, không mất nhiều công sức thiết kế dây.
- Cài đặt linh hoạt: công nghệ không dây cho phép mạng kết nối đến mọi nơi, bao gồm cả những đối với mạng LAN có dây là không thể.
- Hạ thấp chi phí triển khai: mặc dù đầu tư ban đầu về phần cứng có thể cao hơn mạng có dây, tuy nhiên xét chi phí tổng thể và chi phí theo tuổi thọ có thể thấp hơn đáng kể. Về lâu dài, WLAN sẽ đem lại lợi ích rất lớn trong các môi trường động yêu cầu sự di chuyển và thay đổi nhiều.
- Khả năng mở rộng: hệ thống WLAN có thể cấu hình trong nhiều mô hình để đáp ứng các ứng dụng và cấu hình đặc thù dễ dàng thay đổi và phạm vi từ mạng điểm - điểm xây dựng cho số nhỏ người dùng đến các mạng phối hợp với hàng ngàn người dùng cho phép chuyển vùng trên phạm vi rộng.

Sự hấp dẫn của WLAN là tính linh hoạt của chúng. Chúng có thể mở rộng mở rộng truy cập tới các mạng cục bộ, như Intranet, cũng như hỗ trợ sự truy nhập băng rộng tới Internet tại các Hotspot. WLAN có thể cung cấp kết nối không dây nhanh chóng và dễ dàng tới các máy tính, các máy móc hay các hệ thống trong một khu vực, nơi mà các hệ thống cơ sở hạ tầng truyền thông cố định không tồn tại hoặc sự truy nhập như vậy là không được phép. Người dùng có thể cố định hoặc di động hoặc thậm chí có thể đang ngồi trên một phương tiện chuyển động.

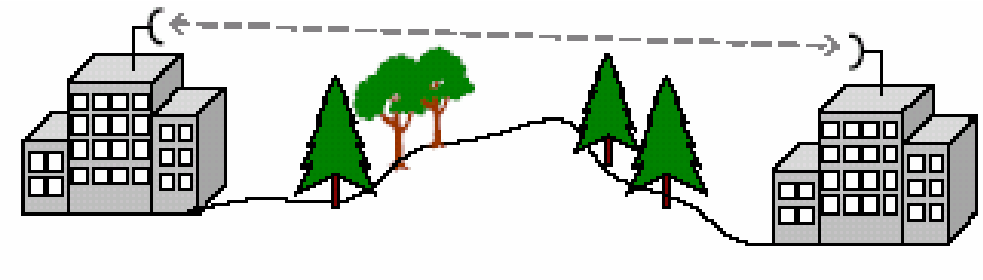
Một vài hình vẽ sau sẽ đưa ra cho ta cái nhìn tổng quan về khả năng ứng dụng của WLAN:

Khả năng sử dụng WLAN để mở rộng mạng hữu tuyến thông thường, với tốc độ cao và tiện lợi trong truy nhập mạng

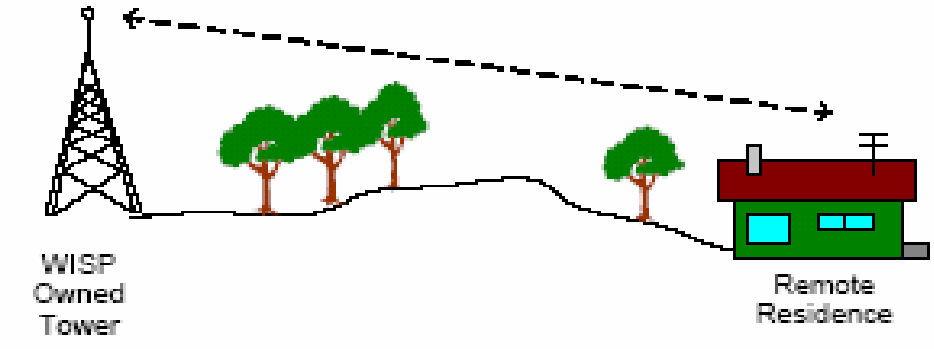


Hình II.1.1 - WLAN kết nối mạng hữu tuyến

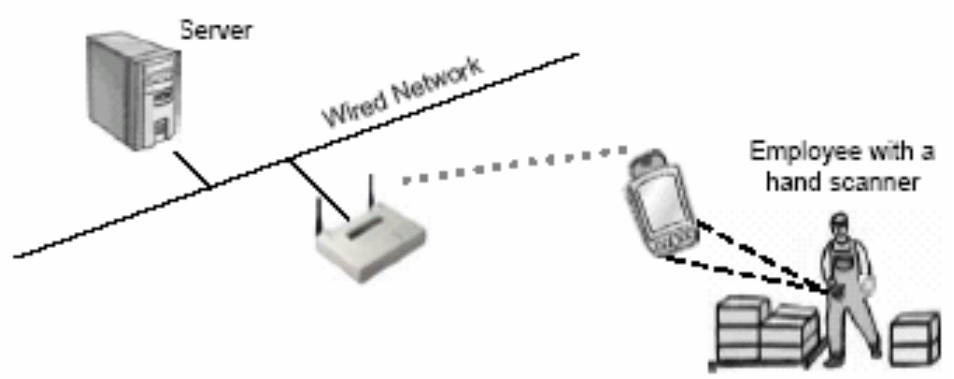
Khả năng đơn giản hóa việc kết nối mạng giữa hai tòa nhà mà giữa chúng là địa hình phức tạp khó thi công đối với mạng thông thường



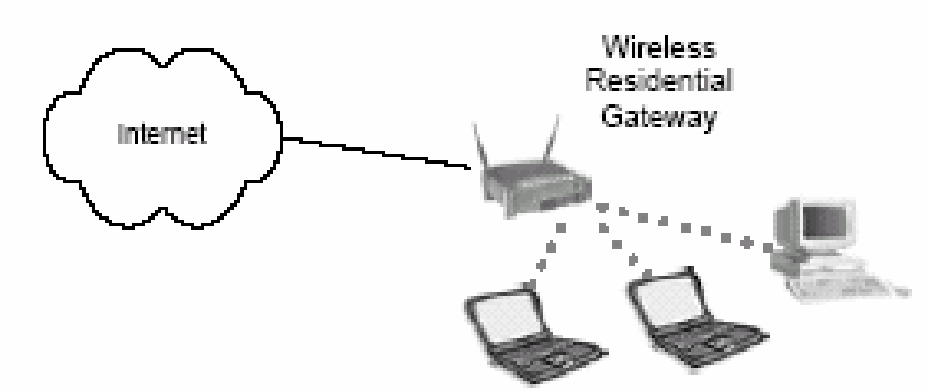
hay các khu vực có địa hình lòng chảo vẫn có thể truy cập mạng bình thường như các nơi khác



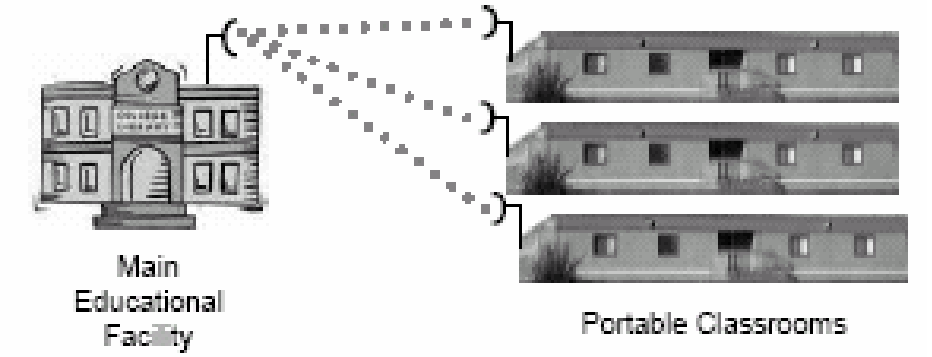
và sự tiện lợi trong việc truy cập mạng mà vẫn có thể di chuyển



Từ các văn phòng, nhà riêng



đến các khu lớn hơn nhiều như các trường đại học, các khu chung cư đều có thể truy cập mạng với tốc độ cao, mà việc triển khai lại đơn giản



1.2 Một số ứng dụng thực tế của WLAN tại Việt Nam

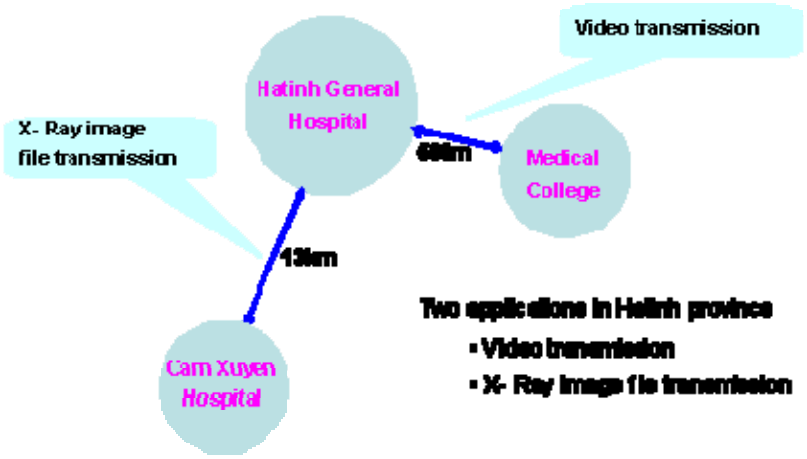
1.2.1 Ứng dụng trong Wireless LAN Telemedicine

Hệ thống tư vấn phẫu thuật trực tuyến (Telemedicine), một hệ thống y tế mới, hiện đại đã được triển khai ở Việt Nam. Mới đây, hệ thống Telemedicine đã chính thức được khai trương vào ngày 5/5/2005; đồng thời diễn ra cuộc phẫu thuật qua cầu truyền hình tại hai bệnh viện Việt Đức (Hà Nội) và bệnh viện Việt Tiệp (Hải Phòng). Được biết, giải pháp kỹ thuật sử dụng cho hệ thống điều trị từ xa và các bệnh viện vệ tinh dựa trên nền tảng cơ sở hạ tầng mạng và truyền dẫn của VNPT. Hệ thống mạng và tích

hợp các hệ thống thiết bị kỹ thuật này gồm có: hệ thống thiết bị truyền dẫn quang trực tiếp (telecast); hệ thống thiết bị cung cấp kết nối ADSL, WAN trên mạng NGN; hệ thống mạng LAN kết nối toàn bộ các thiết bị tại mỗi điểm; hệ thống thiết bị IP Camera cung cấp giải pháp hội nghị truyền hình và hệ thống thiết bị số hóa hình ảnh và lưu trữ toàn bộ ca mổ.

Tuy nhiên ở các vùng có địa hình phức tạp và hạ tầng truyền thông chưa phát triển, thì phương án trên không có tính khả thi. Để khắc phục khó khăn này, người ta đã triển khai dự án Telemedicine dựa vào mạng WLAN.

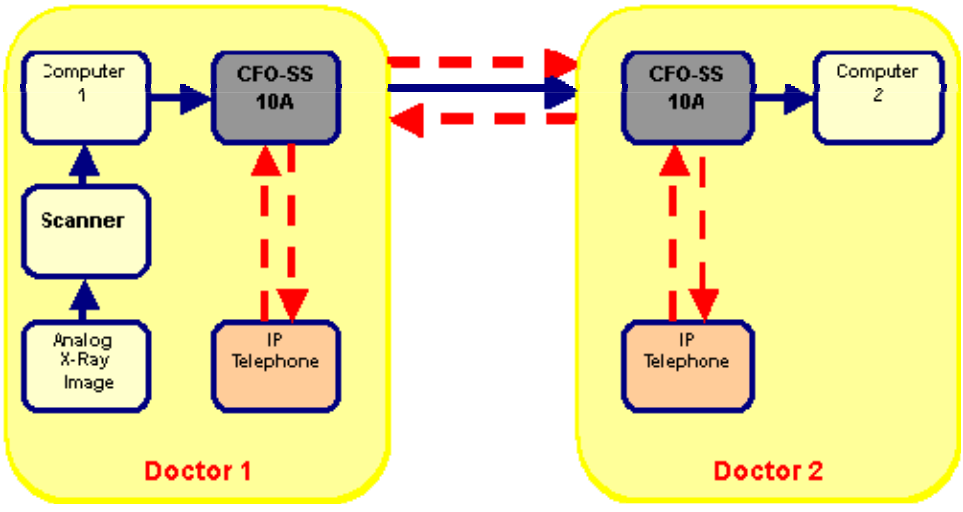
Trên thực tế đã có một hệ thống Telemedicine dựa trên WLAN được triển khai tại tỉnh Hà Tĩnh. Hệ thống này gồm 3 khu vực cơ bản:



Dựa trên công nghệ của WLAN, hệ thống này có thể thực hiện hai công việc:

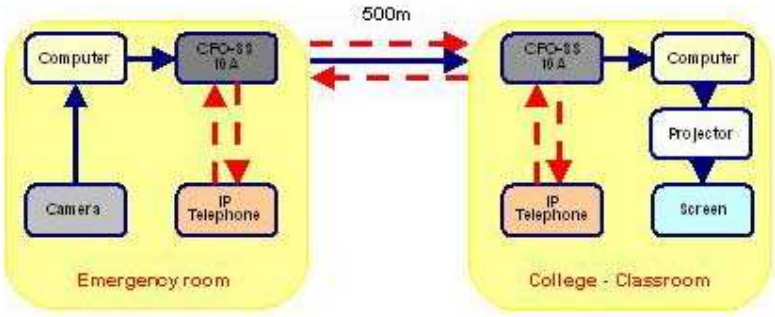
- **Truyền ảnh X-quang**

Ứng dụng này sẽ truyền ảnh X-quang giữa bệnh viện tỉnh Hà Tĩnh và bệnh viện Cẩm Xuyên với khoảng cách 13 km.



- **Truyền Video**

Ứng dụng này được thực hiện giữa bệnh viện tỉnh Hà Tĩnh và trường Đại học Y Hà Tĩnh với khoảng cách 500m. Tín hiệu video sẽ được truyền trực tiếp từ phòng phẫu thuật tới lớp học tại ĐH Y Hà Tĩnh. Các sinh viên tại đây có thể theo dõi từ xa quá trình phẫu thuật của các bác sỹ.



1.2.2 Hệ thống WiFi VNN

Định nghĩa: "Wi-Fi là dịch vụ truy nhập Internet tốc độ cao bằng công nghệ không dây theo chuẩn 802.11 IEEE".

Công nghệ sử dụng: IEEE 802.11b

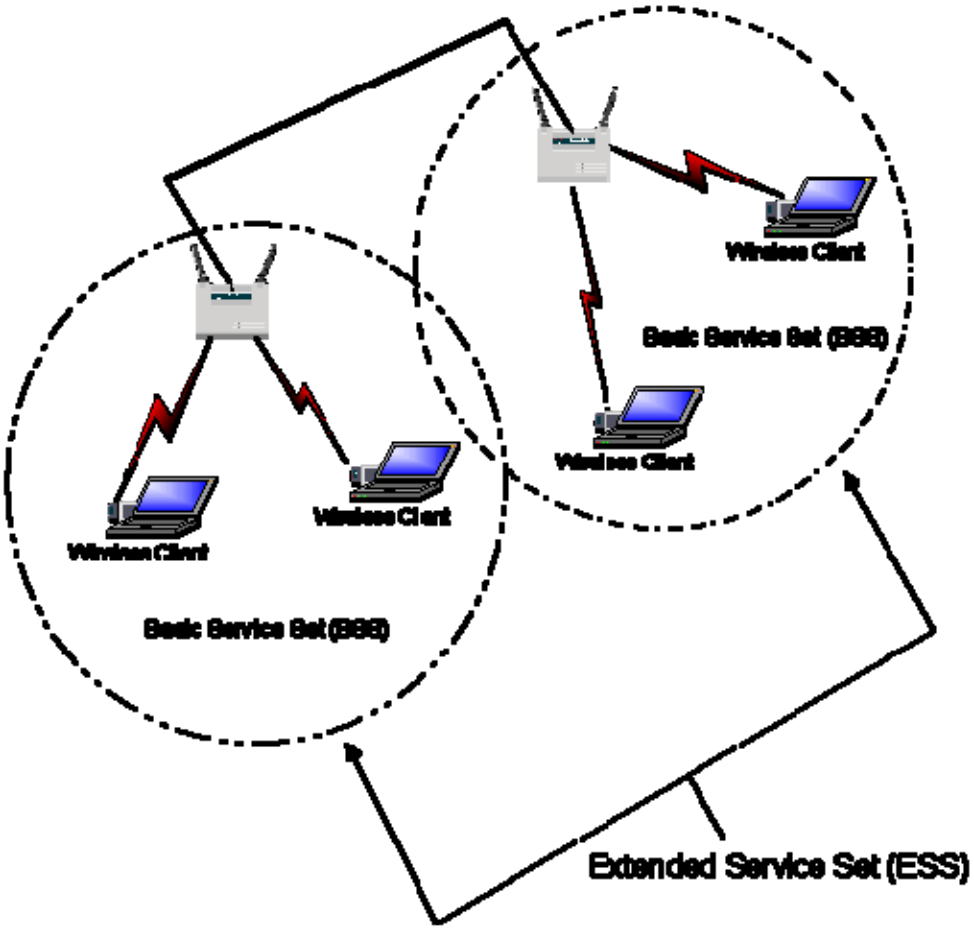
- Kích thước phủ sóng của mỗi HOTSPOT: < 300m.
- Tần số: 2,4 GHz (giải IMS), công suất phát: ≤ 100mW, độ rộng băng thông: 22MHz.
- Tốc độ: 1Mbps, 2Mbps, 5.5 Mbps, 11 Mbps.
- Bảo mật: WEP.
- Hệ quản lý: Radius (AAA).
- Kỹ thuật trải phổ: DSSS, FHSS.
- Tính cước: theo thời gian, volume, hoặc flate rate.
- Đăng ký: theo account.

2. CÁC MÔ HÌNH MẠNG CƠ BẢN

WLAN thường có hai kiểu Mô hình mạng:

- Mô hình cơ sở (Infrastructure network)
- Mô hình độc lập (Independent network - Ad hoc network)

2.1 Mô hình cơ sở (Infrastructure network)



Hình II-2.1 - Mô hình cơ sở

Luôn phải có 1 AP và một số client. Mỗi client này phải có card giao tiếp vô tuyến (adapter) để kết nối với AP.

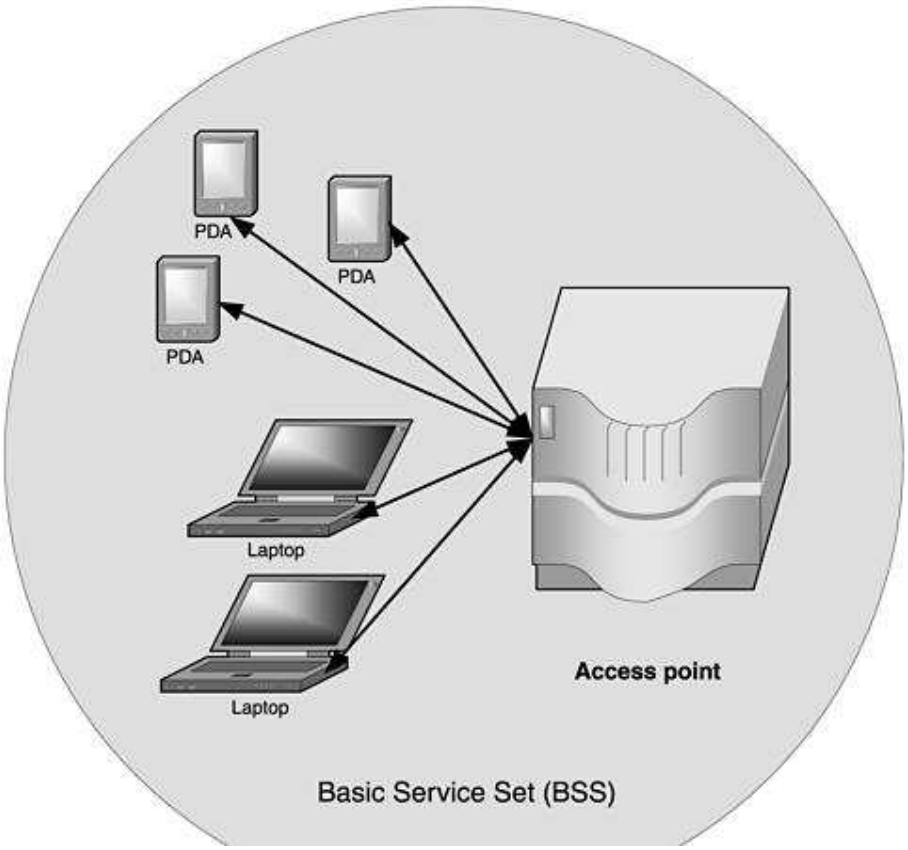
Trong chế độ cơ sở (Infrastructure mode or Infrastructure network) lại có thể chia làm 2 loại (theo số lượng AP):

- Mô hình cơ sở - Basic Service Set (BSS)
- Mô hình mở rộng - Extended Service Set (ESS)

2.1.1 Tập hợp dịch vụ cơ bản (BSS - Basic Service Set)

BSS bao gồm:

- Một AP kết nối với 1 mạng có dây (wired network)
- Một số thiết bị khách (a set of stations)



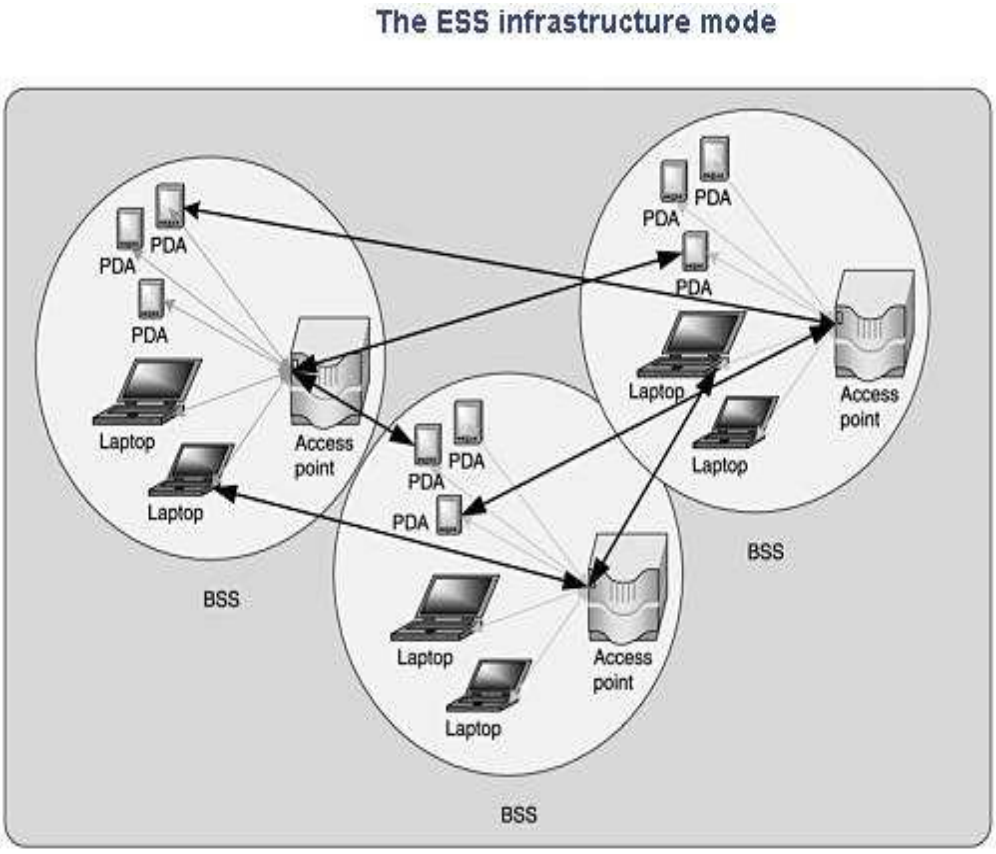
Hình II-2.2 - Tập các dịch vụ cơ sở

Vì BSS hoạt động ở chế độ cơ sở nên bắt buộc phải sử dụng AP và tất cả quá trình trao đổi dữ liệu vô tuyến (wireless traffic) đều thông qua AP. Và không có trao đổi trực tiếp giữa các client. Tức là mỗi thiết bị vô tuyến (wireless client) phải sử dụng AP để giao tiếp với nhau và với các máy tính trong mạng có dây (wired host).

2.1.2 Tập hợp các dịch vụ mở rộng (ESS Extended Service Set)

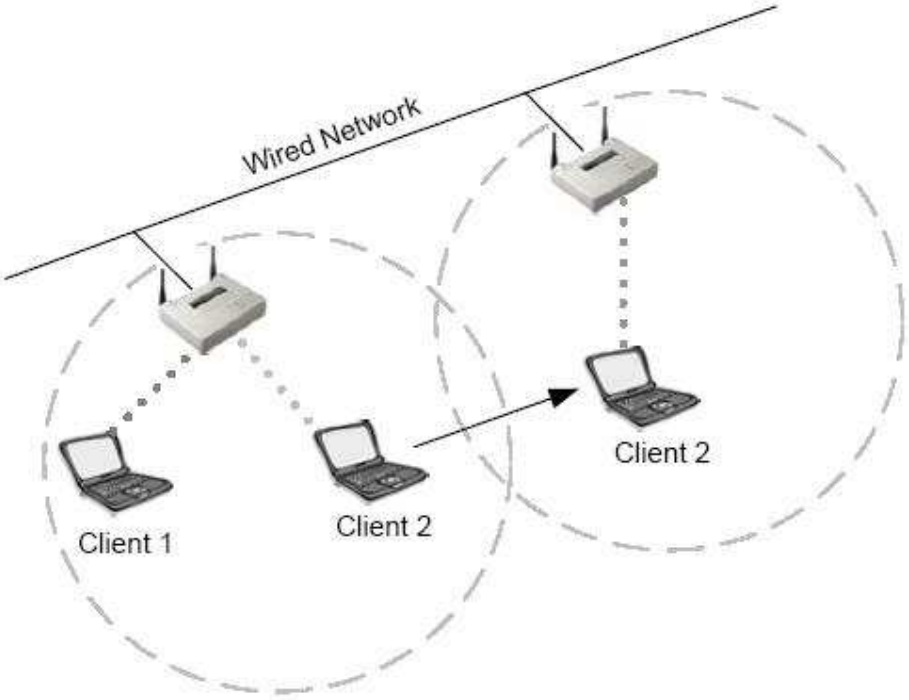
Mô hình mở rộng (ESS) là sự liên kết của 2 hay nhiều mô hình cơ sở bởi một hệ thống phân tán (distribution system). Hệ thống phân tán này có thể là hữu tuyến (wired) như LAN hay WAN, hoặc là vô tuyến (wireless).

Từ đó ta cũng có thể thấy một mô hình mở rộng cần có ít nhất 2 AP cùng hoạt động ở chế độ cơ sở.



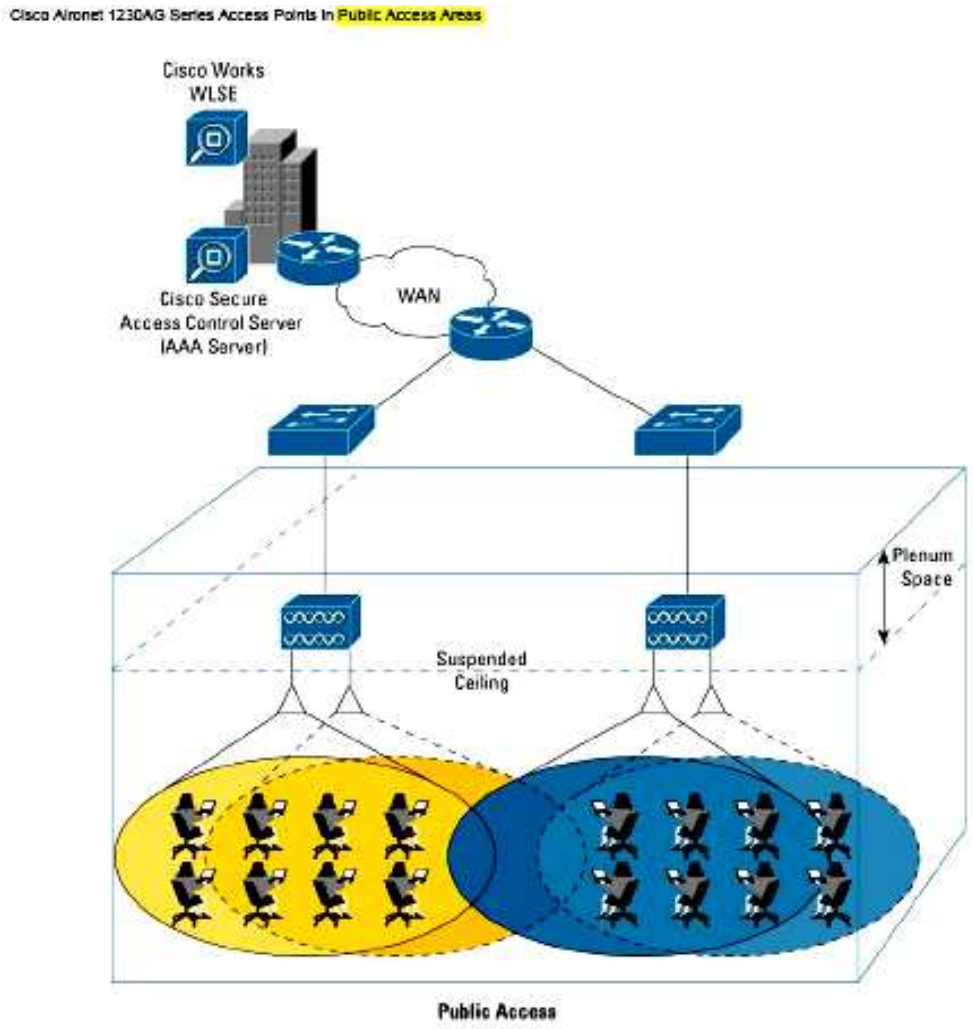
Hình II-2.3 - Tập các dịch vụ mở rộng

Mỗi mô hình mở rộng thường bao gồm nhiều cell (BSS), nên có hiện tượng bao phủ (coverage overlap) giữa các cell. Như trong sơ đồ dưới đây, 2 vùng phủ sóng có phần đan xen vào nhau.



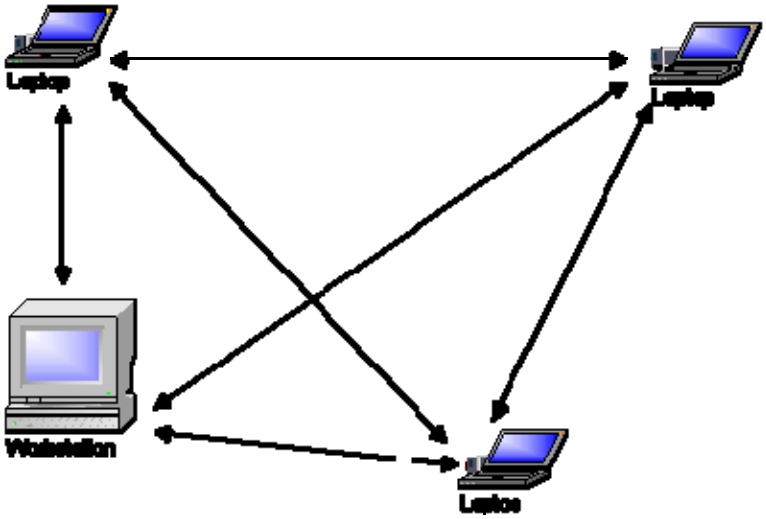
Hình II-2.4 - Sự đan xen của hai vùng phủ sóng

Và đây chính là mô hình hay gặp trong thực tế. Bởi lẽ, WLAN thường là giải pháp triển khai bổ sung thêm vào hệ thống mạng hiện có



Hình II-2.5 - Mô hình mạng thực tế

2.2 Mô hình Adhoc độc lập (Independent network)



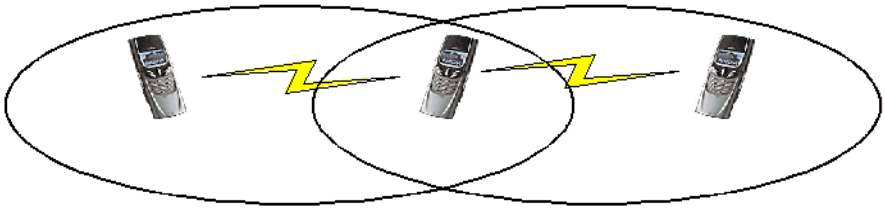
Hình II-2.6 - Mô hình mạng Ad-hoc

Mô hình Adhoc hay còn được gọi là MANET (Mobile Ad-hoc Network) có thể tạm dịch là Mô hình độc lập (Independent network)

Đặc điểm nổi bật của mô hình này là không có AP. Do đó các client tương tác trực tiếp với nhau. Đây là một mô hình mạng ngang hàng (peer- to – peer network), tức là mỗi nút mạng tự dẫn đường để truyền tin.

Trong mô hình này, mỗi nút mạng sẽ hoạt động giống như một máy chủ (host) và đồng thời như một bộ trọn đường (router). Điều này có nghĩa là mỗi nút trong mạng MANET sẽ đóng vai trò như một trạm chuyển tiếp, được dùng để chuyển các gói tin tới các nút lân cận cho đến khi gói tin đến được nút đích. Nói một cách khác sự tồn tại của MANET phụ thuộc vào sự kết hợp giữa các nút thành viên.

Hoạt động của AdHoc có thể tóm tắt như sau: các nút sẽ tuần tự chuyển tiếp các gói tin đó đến nút tiếp theo trong mạng cho đến khi gói tin đến được nút đích. Như vậy trong mô hình này một gói tin có thể sẽ phải truyền qua nhiều nút trước khi tới được nút đích.



Hình II-2.7 – Hoạt động truyền tin trong mạng Ad-hoc

Hình trên mô phỏng một mạng không dây đơn giản theo mô hình ad-hoc bao gồm có 3 nút mạng. Xét trường hợp nút nguồn S muốn kết nối tới nút đích D, tuy nhiên giữa S và D không có một kết nối trực tiếp khi đó nút R sẽ được sử dụng như một nút trung gian. Nút R sẽ có vai trò chuyển tiếp các gói tin trong kênh kết nối giữa S và D. Do đó mặc dù R có thể là một máy chủ nhưng trong trường hợp này nó hoạt động tương tự như một bộ chọn đường (router).

3. HOẠT ĐỘNG CỦA CÁC CHUẨN LIÊN QUAN

3.1 Các băng tần ISM

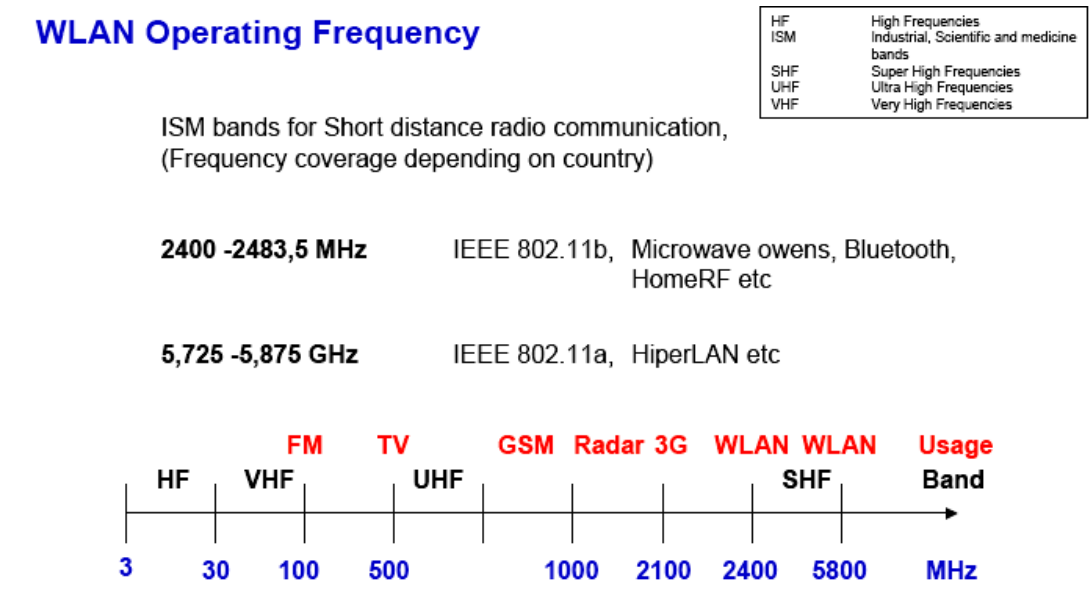
Trong phần trên, ta có nói sơ lược về các thông số liên quan tới băng thông của các chuẩn. Các băng thông trên là do FCC quy định.

FCC (Federal Communication Commission) hội đồng truyền thông thành lập từ năm 1934 tại Mỹ. Nhiệm vụ của cơ quan này là ban hành các quy định về truyền thông trong nước Mỹ và quốc tế với các lĩnh vực: radio, TV, vệ tinh, cáp (cable), mạng máy tính …

Đối với mạng cục bộ vô tuyến, FCC quy định về:

- Tần số sóng vô tuyến cho phép đối với WLAN (để tránh xung đột với sóng của các hệ thống truyền thông khác)
- Năng lượng (công suất) cho phép
- Các thiết bị WLAN nào được hoạt động và 1 số quy định khác kèm theo

Điều mà ta quan tâm chủ yếu ở đây là các tần số sóng vô tuyến cho phép, hay nói khác đi là các băng tần.



FCC quy định 2 loại băng tần: ISM và UNII dành cho WLAN.

Sau đây ta sẽ tìm hiểu cụ thể 2 loại băng tần này đối với các mạng vô tuyến

3.1.1 Băng tần ISM (ISM bands)

ISM bands là viết tắt của Industrial - Scientific - Medical bands.

Có 3 loại băng tần ISM dành cho mạng vô tuyến:

- 900 MHz

- 2.4 GHz
- 5.8 GHz

Băng tần ISM 900 MHz

- Miền tần số theo quy định của FCC: 900 MHZ – 928 MHz (915 MHz ± 13 MHz) Tức là ISM 900 MHz có thông lượng 13 MHz.
- WLAN đã từng hoạt động ở băng tần này. Nhưng bây giờ không sử dụng nữa, vì thiết bị đắt mà tốc độ lại không cao (chỉ 1Mbps)
- Tuy nhiên băng tần ISM 900 MHz vẫn được sử dụng ở 1 số thiết bị không dây: wireless home phone, wireless camera system..

Băng tần ISM 2.4 GHz

- Được sử dụng rộng rãi hiện nay với các thiết bị theo chuẩn 802.11, 802.11b, 802.11g…
- Miền tần số theo FCC quy định: 2.4 GHz – 2.5 GHz (2.45 GHz ± 50 MHz)
- Tuy nhiên các thiết bị thực tế chỉ sử dụng miền tần số: 2.4 GHz – 2.4835 GHz

Băng tần ISM 5.8 GHz

- Miền tần số quy định: 5.725 GHz – 5.875 GHz
- Hiện nay không được sử dụng với WLAN

3.1.2 Băng tần UNII (UNII bands)

UNII là viết tắt của Unlicensed National Information Infrastructure bands

Băng tần UNII là các băng tần từ 5 GHz trở lên và cách nhau 100 MHz (độ rộng băng tần - wide band = 100 MHz).

Có 3 loại băng tần UNII

- Băng tần thấp (Lower bands)

  Miền tần số: 5.15 GHz ÷ 5.25 GHz

  Công suất cực đại: 50 mW. Nhưng trong thực tế các thiết bị WLAN (chuẩn 802.11a) thường công suất chỉ còn 40 mW.

- Băng tần trung bình (Midle bands)

  Miền tần số: 5.25 GHz ÷ 5.35 GHz

  Công suất cực đại: 250 mW (theo quy định của FCC). Nhưng thực tế các chuẩn của IEEE chỉ có 200 mW

- Băng tần cao (Upper bands)

  Miền tần số: 5.725 GHz ÷ 5.825 GHz

  Công suất cực đại: 1 W (FCC) , 800 mW (IEEE)

3.2 Các chuẩn 802.11 (IEEE 802.11 family)

Các chuẩn này do Viện kỹ sư Điện và Điện tử - IEEE (Institute of Electrical and Electronics Engineers) tạo ra. IEEE là một tổ chức vô cùng quen thuộc trong lĩnh vực mạng và viễn thông. Trước đó họ đã ban hành một loạt các chuẩn nổi tiếng như: chuẩn 802.1 về kiến trúc LAN, 802.3 - Ethernet, 802.5 – Token Ring, 802.8 – Cáp quang (Fiber Optics), 802.14 – Cable TV… Và bây giờ là họ tiêu chuẩn 802.11 dành cho WLAN.

Chúng ta sẽ lần lượt xem xét các chuẩn mà IEEE ban bố theo thứ tự thời gian

- 802.11
- 802.11b
- 802.11a
- 802.11g

3.2.1 IEEE 802.11

Đây là chuẩn WLAN đầu tiên được công bố

Chuẩn này bao gồm các hệ thống trải phổ trực tiếp (DSSS) và trải phổ dịch tần (FHSS), có tốc độ truyền dữ liệu 1 Mbps và 2 Mbps. Hoạt động ở băng tần ISM 2.4 GHz (2.4 GHz ± 2.4835 GHz)

3.2.2 IEEE 802.11b

IEEE 802.11b được biết đến với tên thương mại là Wi-Fi.

IEEE 802.11b chỉ sử dụng các hệ thống trải phổ trực tiếp (DSSS), không sử dụng trải phổ dịch tần (FHSS).

Tốc độ truyền dữ liệu (data rate): 1 Mbps, 2 Mbps, 5.5 Mbps và tối đa là 11 Mbps.

Hoạt động ở băng tần ISM 2.4 GHz (2.4 GHz ± 2.4835 GHz)

Các thiết bị của chuẩn 802.11b thì cũng là 802.11. Tức là chúng tương thích với các thiết bị 802.11 trước đó. Điều này giúp giảm bớt giá thành khi nâng cấp.

3.2.3 IEEE 802.11a

Hoạt động ở băng tần UNII 5 GHz. Trong khi 2 chuẩn trước đó đều hoạt động ở băng tần ISM 2.4 GHz. Do đó các thiết bị chuẩn 802.11a sẽ không tương thích với các thiết bị theo chuẩn 802.11 và 802.11b

Tốc độ truyền dữ liệu (data rate): 6 Mbps, 9 Mbps, 12 Mbps, 18 Mbps, 36 Mbps, 48 Mbps và tối đa là 54 Mbps.

3.2.4 IEEE 802.11g

Mục đích của IEEE khi cho ra đời chuẩn 802.11g là muốn: có được tốc độ như 802.11a. Đồng thời lại phải tương thích với 802.11 và 802.11b (tức là cũng sử dụng băng tần ISM 2.4 GHz).

Để đạt được 2 mục đích này, 802.11g sử dụng kỹ thuật điều chế ODFM – Orthogonal Frequency Division Multiplexing (Chia tần đa trực giao).

Đây là chuẩn được IEEE chính thức ban hành vào 6/2003 và vẫn đang được hoàn thiện.

Được giới thiệu vào năm 2003, 802.11g kết hợp một số các đặc điểm tốt nhất của hai chuẩn trước đây. Nói một cách chính xác hơn thì đây là sự mở rộng và cải tiến của 802.11b, minh chứng là :

- Khoảng cách làm việc là 100 ÷ 150 feet.
- Tương thích với các thiết bị sử dụng chuẩn 802.11b.
- Sử dụng băng tần 2,4GHz.
- Bị xuyên nhiễu khi sử dụng cùng không gian với các thiết bị trong cùng dải tần.

Cũng như chuẩn "A", chuẩn 802.11g có thể truyền dữ liệu với tốc độ từ 15Mbps ÷ 20Mbps. Và nếu bạn đã từng sử dụng công nghệ 802.11b thì hãy chú ý đến sản phẩm bạn sử dụng có được hỗ trợ việc nâng cấp firmware lên 802.11g hay không? Nếu được hỗ trợ thì hiển nhiên là bạn có khả năng tận hưởng những cải tiến của 802.11g bằng việc nâng cấp firmware cho các sản phẩm của mình.

3.2.5 Một số chuẩn khác trong họ IEEE 802.11

IEEE 802.11n

Liên minh WWiSE (worldwide spectrum efficiency) - bao gồm các công ty Airgo Networks, Bermai, Broadcom, Conexant Systems, STMicroelectronics và Texas Instruments - cho biết công nghệ WI-FI mới đang được nhóm thảo luận 802.11n của Viện Kỹ thuật Điện và Điện tử (IEEE) xem xét. Đây là bộ phận giám sát một chuẩn Wi-Fi thế hệ kế tiếp có khả năng duy trì tốc độ trao đổi dữ liệu không dây vượt mức 100Mbps.

Chuẩn WI-FI đề xuất trên dựa vào công nghệ MIMO-OFDM (multiple input, multiple output-orthogonal frequency division multiplexing), cung cấp tốc độ cao hơn bằng cách sử dụng hai ăng-ten ở mỗi đầu của tín hiệu thay vì một ăng-ten ở mỗi đầu như hiện nay.

Đề xuất của WWiSE, vốn là một trong hai lựa chọn cạnh tranh cho chuẩn 802.11n, được thiết kế để có thể triển khai trên toàn cầu và tương thích ngược với các chuẩn WI-FI hiện tại như 802.11b, .11a và .11g. Trong tuyên bố về bản quyền sáng chế của mình, các công ty thành viên WWiSE sẽ không tính phí bản quyền đối với nhà phát triển công nghệ 802.11n, một nỗ lực nhằm giảm chi phí cho các nhà phát triển, nhà sản xuất và người tiêu dùng.

Công nghệ MIMO sẽ là thành phần cốt yếu của chuẩn 802.11n, cung cấp phạm vi phủ sóng WLAN ổn định hơn với tỷ lệ truyền dữ liệu siêu nhanh: 'Nó sẽ cho phép người dùng thực hiện nhiều công việc hơn với WI-FI, đặc biệt trong các ứng dụng đa phương tiện. Với các khách hàng doanh nghiệp, chuẩn không dây mới này cuối cùng sẽ vượt qua tốc độ của mạng nội bộ truyền thống' .Các sản phẩm 'trước n' (Pre-n) trong chuẩn WI-FI đang được đưa ra để đáp ứng nhu cầu của các nhà sản xuất thiết bị mạng WLAN thế hệ kế tiếp.

Hai ăng-ten ở mỗi đầu thiết bị giúp chuẩn 802.11n tăng tốc độ không dây lên gấp bốn lần
Bằng cách sử dụng cùng định dạng kênh 20MHz mà hàng triệu thiết bị WI-FI hiện

đang sử dụng, chuẩn mới của WWiSE cung cấp khả năng tương thích với nền tảng đã cài đặt trước, đồng thời nâng cao hiệu suất của các mạng WLAN trong dải tần số RF được phân bổ.

WWiSE cho biết công nghệ mới có thể đạt tỷ lệ truyền dữ liệu tối đa lên tới 135Mbps trong cấu hình bắt buộc tối thiểu two-by-two, và tỷ lệ này có thể lên tới 540Mbps qua một cấu trúc MIMO four-by-four và độ rộng kênh truyền 40MHz.

IEEE 802.11i

Trong khi chuẩn 801.11n đang triển khai mạnh mẽ, thì phiên bản hiện hành của chuẩn 802.11i được xét xem có đủ độ tin cậy để trở thành một chuẩn chung cho cả ngành hay không?

Theo chủ tịch hội đồng xét duyệt chuẩn IEEE 802.11 thì "Phiên bản này đã được uỷ ban xét duyệt nghiên cứu". Đặc trưng của 802.11i là dữ liệu truyền qua các mạng không dây sẽ được mã hóa để đảm bảo không thể bị đột nhập hoặc can thiệp bởi các yếu tố bên ngoài. Các phương pháp bảo mật trước đây, điển hình là Wired Equivalent Privacy (WEP), rất dễ bị tin tặc tấn công vì chứa nhiều điểm yếu. Chính vì vậy từ đầu năm 2003, một phương pháp bảo mật mới có tên 'WI-FI Protected Access" (WPA) đã được phát triển và đưa vào sử dụng. Tiếp sau phương pháp bảo mật này, IEEE đã đưa ra chuẩn 802.11i với mục đích chủ yếu là tăng cường an ninh cho mạng không dây.

Trên thực tế, khá nhiều tính năng mới của chuẩn 802.11i đã và đang được sử dụng trong các thiết bị WI-FI hiện nay. Điển hình trong số đó là chuẩn mã hoá cao cấp AES, một chuẩn mã hoá cực mạnh hỗ trợ khóa 128-bit, 129-bit và 256-bit.

IEEE 802.11c

IEEE 802.11c hỗ trợ các khung (frame) thông tin của 802.11.

IEEE 802.11d

Hỗ trợ các khung thông tin của 802.11 nhưng tuân theo những tiêu chuẩn mới.

IEEE 802.11e

Nâng cao QoS ở lớp MAC.

Đây là một chuẩn IEEE được đề xuất nhằm xác định cơ chế QoS (Quality of Service) của mạng không dây để hỗ trợ các ứng dụng của băng rộng như giọng nói và hình ảnh video.

IEEE 802.11f

Inter Access Point Protocol

IEEE 802.11h

Có thêm tính năng lựa chọn kênh tự động, Dynamic Channel Selection (DCS) và điều khiển công suất truyền dẫn (Transmit Power Control).

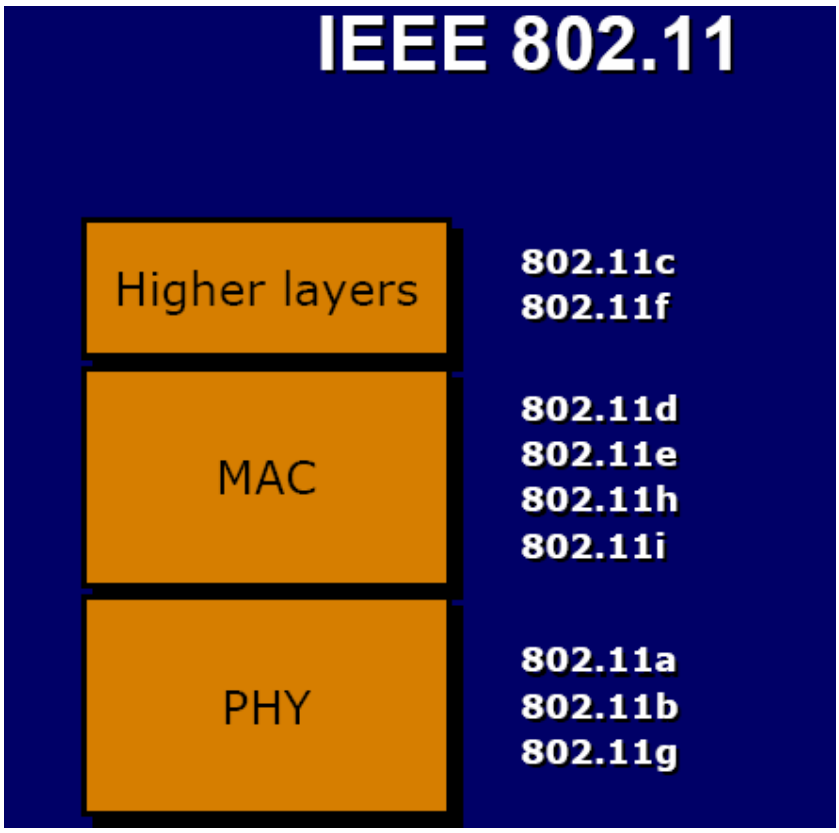
IEEE 802.1x

Một chuẩn mới được cập nhật và thực hiện, nó cung cấp sự điều khiển truy cập mạng trên cổng cơ sở. Mặc dù lúc đầu IEEE thiết kế 802.1x cho thông tin hữu tuyến, nhưng đã được áp dụng cho WLANs để cung cấp một vài sự bảo mật cần thiết. Lợi ích chính

của 802.1x đối với WLANs là nó cung cấp sự chứng thực lẫn nhau giữa Authentication Server và client.

IEEE 802.11j

Là chuẩn thống nhất toàn cầu cho các tiêu chuẩn: IEEE, ETSI, HiperLAN2, ARIB, HiSWANa.

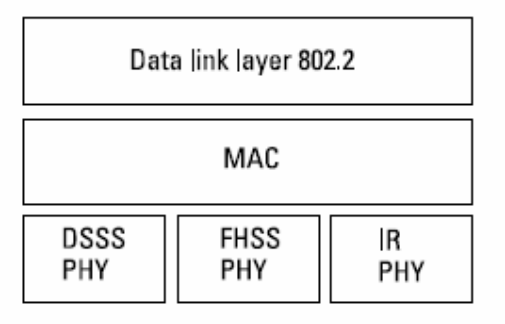


Hình II-3.1 - Các chuẩn trong họ 802.11

4. MÔ HÌNH CẤU TRÚC CỦA MẠNG WLAN

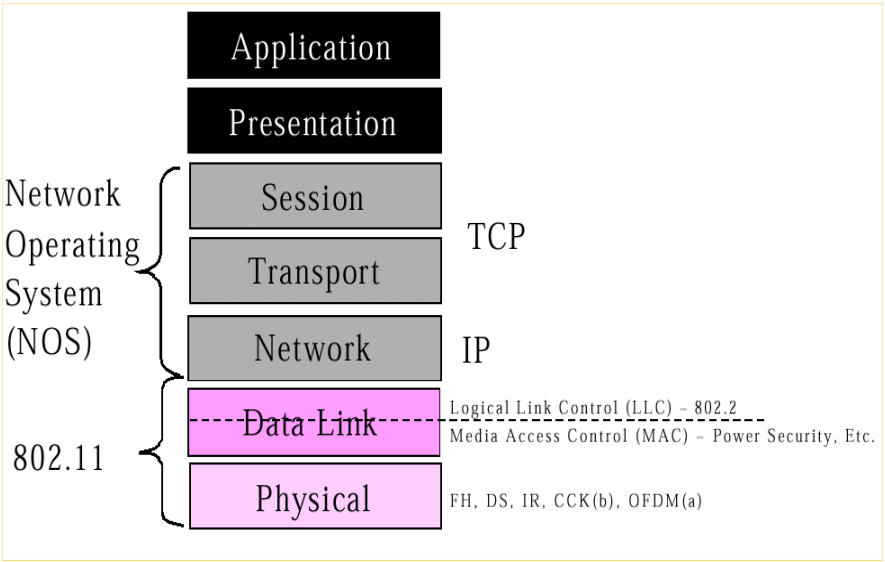
4.1 Tầng vật lý (PHY layer)

Họ IEEE 802.11 sử dụng 3 lớp vật lý khác nhau trong các bản tiêu chuẩn của mình. Nhưng tất cả đều nằm dưới 1 lớp MAC duy nhất.



Hình II-4.1 - Cấu trúc tầng vật lý

Còn lại các tầng trên là chung



Hình II-4.2 - So sánh với mô hình OSI

Lớp vật lý được chia thành hai phân lớp: PLCP (Physical Layer Convergence Procedure Sublayer - phân lớp thủ tục hội tụ vật lý) và PMD (Physical Media Dependent Sublayer - phân lớp vật lý trung gian phụ thuộc).

PLCP đưa các chức năng của hệ thống vật lý trung gian phụ thuộc - vào các dịch vụ của lớp vật lý. PLCP xác định một cách thức sắp xếp các đơn vị dữ liệu dịch vụ của phân lớp PHY 802.11 (PSDU – PHY sublayer Service Data Units) thành một khuôn mẫu phù hợp cho việc truyền và nhận dữ liệu và quản lý thông tin giữa hai hay nhiều

trạm, sử dụng hệ thống vật lý trung gian phụ thuộc tương ứng. Điều này cho phép lớp MAC hoạt động ít phụ thuộc nhất vào phân lớp PMD.

PMD xác định các tính chất và cách thức truyền và nhận dữ liệu thông qua một thiết bị wireless trung gian giữa hai hay nhiều trạm, các trạm sử dụng cùng một hệ thống điều biên.

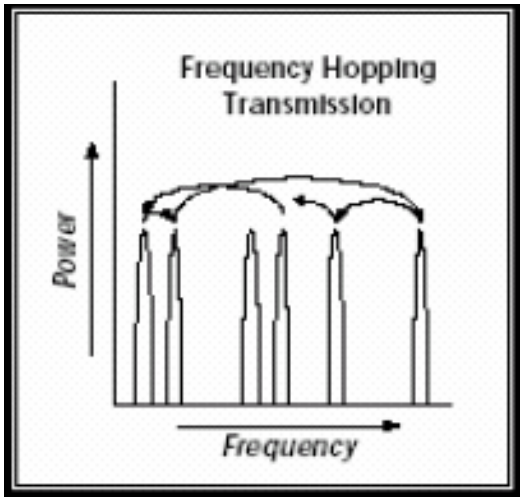
Cho tới nay, IEEE 802.11 định rõ năm dạng của tầng vật lý:

- Trải phổ dịch tần (FHSS - Frequency Hopping Spread Spectrum)
- Trải phổ trực tiếp (DSSS - Direct Sequence Spread Spectrum)
- Hồng ngoại (IR - Infrared)
- Trải phổ trực tiếp tốc độ cao (HR/DSSS - High Rate Direct Sequence Spread Spectrum)
- Kết hợp tần số phân chia trực giao (OFDM - Orthogonal Frequency Division Multiplexing)

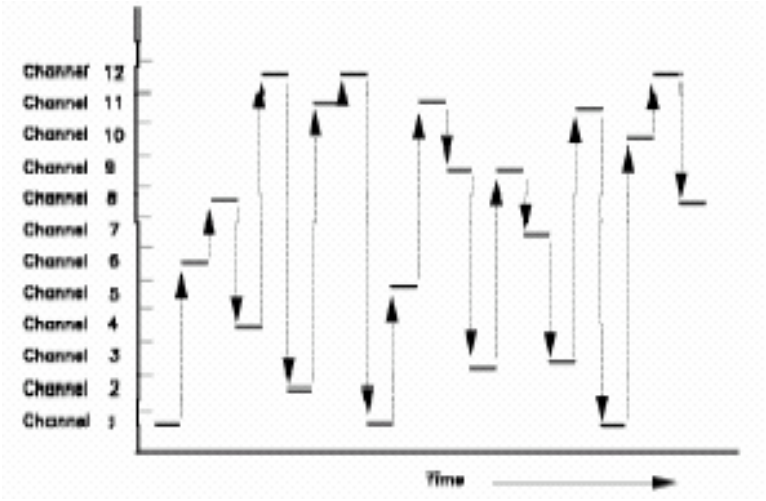
Hai phương pháp cuối cùng được dùng trong mạng WLAN tốc độ cao: IEEE 802.11a sử dụng OFDM, IEEE 802.11b sử dụng HR/DSSS.

4.1.1 Các kỹ thuật trải phổ

Khi tài nguyên vô tuyến ngày càng trở nên cạn kiệt, người ta bắt đầu phải áp dụng kỹ thuật trải phổ nhằm nâng cao hiệu năng sử dụng tần số. Có hai kỹ thuật trải phổ thông dụng nhất hiện nay là FHSS và DSSS. Băng thông cho mỗi CPE sẽ không còn là một dải hẹp mà sẽ là toàn bộ băng tần số, việc xác định CPE thông qua một mã code của mỗi CPE - mã giả ngẫu nhiên (PN sequence).



Hình II-4.3 - FHSS (Frequency Hopping Spread Spectrum)



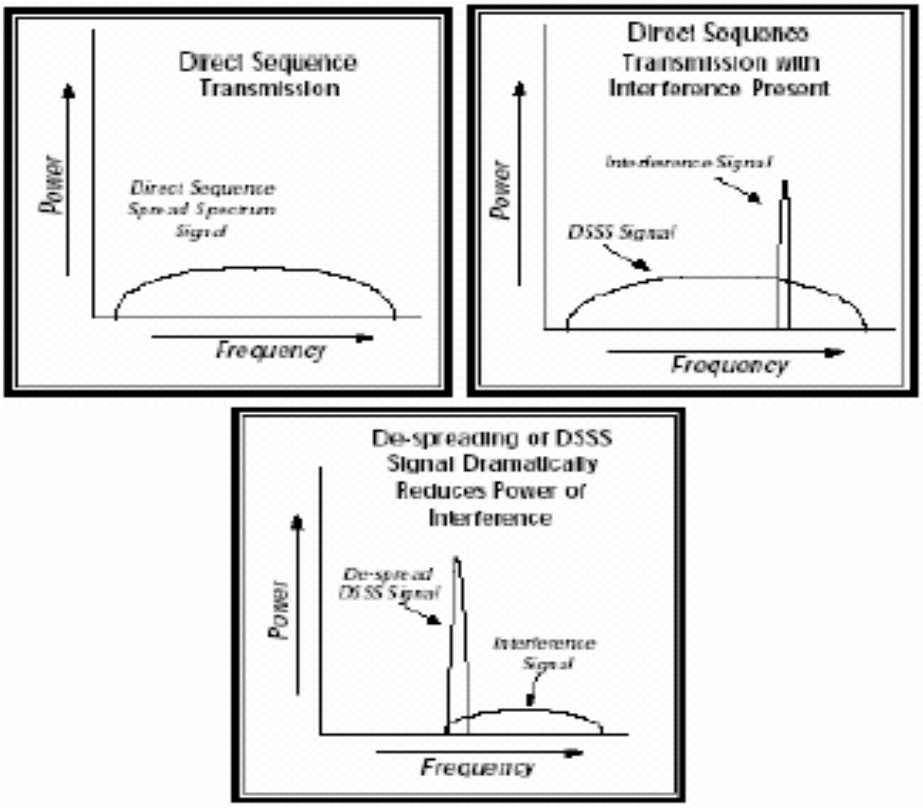
Hình II-4.4 - Nhẩy tần số

Các kênh trong FHSS

Tín hiệu dữ liệu được truyền trên một dải tần rộng bằng kỹ thuật truyền tín hiệu trên những tần số sóng mang khác nhau tại những thời điểm khác nhau. Khoảng cách giữa các tần số sóng mang FHSS được qui định trước, băng thông cho mỗi kênh khoảng 1Mhz, trật tự nhảy tần được xác định bằng một hàm giả ngẫu nhiên. FCC yêu cầu băng thông phải được chia ít nhất thành 75 kênh (subchannel). FHSS radio được giới hạn chỉ gửi một lượng nhỏ dữ liệu trên mỗi kênh trong một chu kỳ thời gian xác định, trước khi nhảy sang kênh tần số kế tiếp trong chuỗi nhảy tần. Chu kỳ thời gian này gọi là dwell time, thường có giá trị khoảng 400 microseconds. Sau mỗi bước nhảy (hop) thiết bị thu phát cần phải thực hiện động bộ lại (resynchronize) với những tần số vô tuyến khác trước khi có thể truyền dữ liệu. Mục đích chủ yếu của việc nhảy tần giả ngẫu nhiên như trên là để tránh hiện tượng giao thoa tín hiệu do kênh dữ liệu không làm việc quá lâu trên một kênh tần số cụ thể nào đó. Giả sử nếu như xảy ra nhiễu giao thoa nghiêm trọng trên một tần số nào đó trong chuỗi nhảy tần thì nó cũng sẽ ảnh hưởng không nhiều đến hệ thống. Bởi quá trình truyền chỉ được thực hiện tại đây trong một khoảng thời gian nhỏ.

DSSS (Direct Sequence Spread Strectrum)

DSSS cũng thực hiện việc trải phổ tín hiệu như trên nhưng theo một kỹ thuật hoàn toàn khác. Băng thông của tín hiệu thay vì được truyền trên một băng hẹp (narrow band) như truyền thông vi ba, sẽ được truyền trên một khoảng tần số lớn hơn bằng kỹ thuật mã hóa giả ngẫu nhiên (Pseudo-Noise sequence).



Quá trình trải và nén phổ trong DSSS

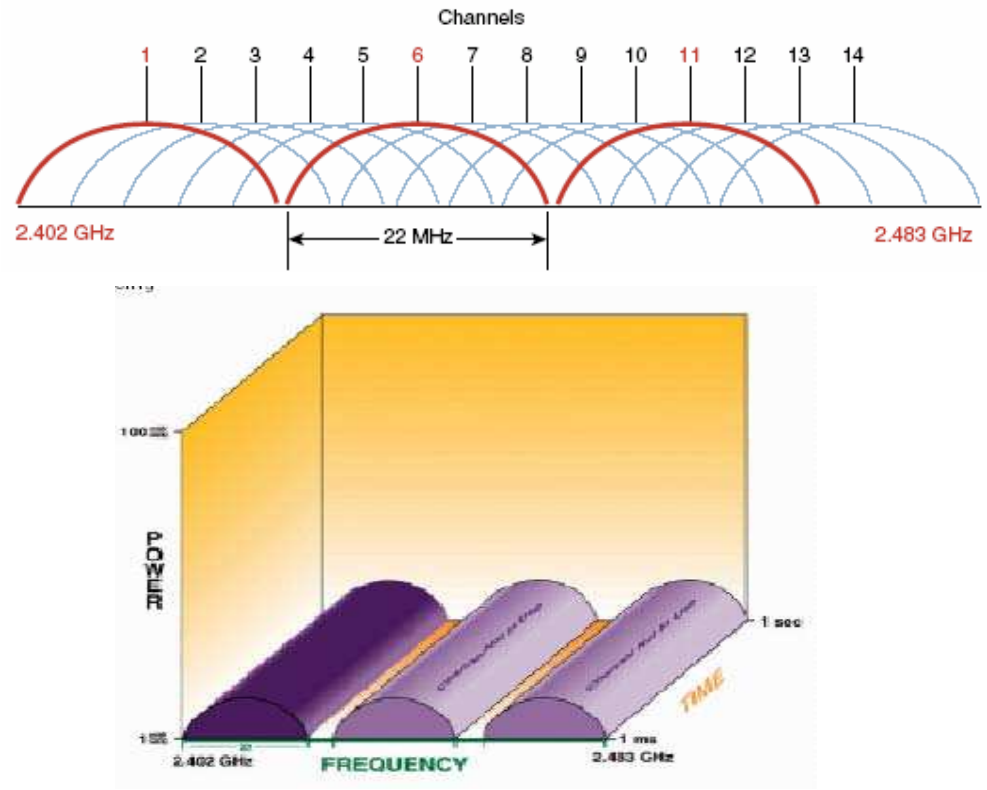
Tín hiệu băng hẹp và tín hiệu trải phổ cùng được phát với một công suất và một dạng thông tin nhưng mật độ phổ công suất (power density) của tín hiệu trải phổ lớn hơn nhiều so với tín hiệu băng hẹp. Tín hiệu dữ liệu kết hợp với chuỗi mã giả ngẫu nhiễn trong quá trình mã hóa sẽ cho ra một tín hiệu với băng thông mở rộng hơn nhiều so với tín hiệu ban đầu nhưng với mức công suất lại thấp hơn. Một ưu điểm nổi bất của kỹ thuật DSSS là khả năng dự phòng dữ liệu. Bên trong tín hiệu DSSS sẽ gộp dự phòng ít nhất 10 dữ liệu nguồn trong cùng một thời gian. Phía thu chỉ cần đảm bảo thu tốt được 1 trong 10 tín hiệu dự phòng trên là đã thành công. Nếu có tín hiệu nhiễu trong băng tần hoạt động của tín hiệu DSSS, tín hiệu nhiễu này có công suất lớn hơn và sẽ được hiểu như là một tín hiệu băng hẹp. Do đó, trong quá trình giải mã tại đầu thu, tín hiệu nhiễu này sẽ được trải phổ và dễ dàng loại bỏ bởi việc sử lý độ lợi (gain processing). Xử lý độ lợi là quá trình làm giảm mật độ phổ công xuất khi tín hiệu được xử lý để truyền và tăng mật độ phổ công suất khi despread, với mục đích chính là làm tăng tỉ số S/N (Signal to Noise ratio).

So sánh FHSS và DSSS

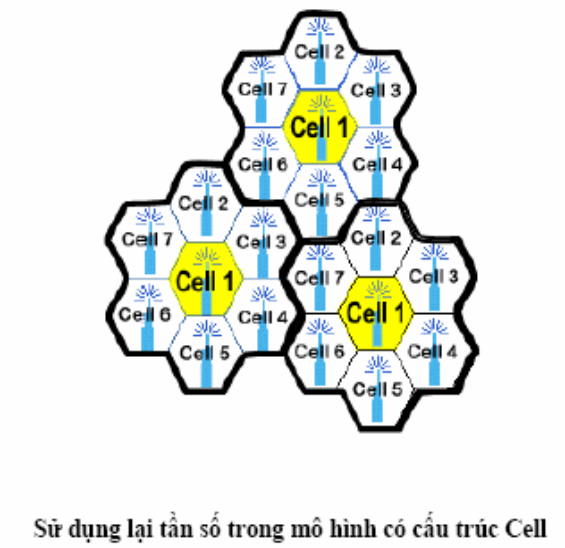
FH không có quá trình xử lý độ lợi do tín hiệu không được trải phổ. Vì thế nó sẽ phải dùng nhiều công xuất hơn để có thể truyền tín hiệu với cùng mức S/N so với tín hiệu DS. Tuy nhiên tại ISM band theo quy định có mức giới hạn công xuất phát, do đó FH không thể được đạt S/N giống như DS. Bên cạnh đó việc dùng FH rất khó khăn trong việc đồng bộ giữa máy phát và thu vì cả thời gian và tần số đều yêu cầu cần phải được đồng bộ. Trong khi DS chỉ cần đồng bộ về thời gian của các chip. Chính vì vậy FH sẽ phải mất nhiều thời gian để tìm tín hiệu hơn, làm tăng độ trễ trong việc truyền dữ liệu hơn so với DS.

Như vậy chúng ta có thể thấy DSSS là kỹ thuật trải phổ có nhiều đặc điểm ưu việt hơn hẳn FHSS.

Theo chuẩn 802.11b, thì sử dụng 14 kênh DS (Direct Sequence) trong dải tần số 2,402GHz – 2,483GHz, mỗi kênh truyền rộng 22MHz, nhưng các kênh chỉ cách nhau 5MHz, vì vậy các kênh cạnh nhau sẽ gây giao thoa lẫn nhau, do đó trong một khu vực người ta bố chí các kênh truyền sao cho miền tần số của chúng không trồng lên nhau, trong hệ thống 14 kênh DS thì chỉ có 3 kênh đảm bảo không chồng lấn, ví dụ như trong hình sau thì các kênh 1, 6, 11 được sử dụng để phát trong một khu vực mà không gây nhiễu giao thoa cho nhau:



Hình II-4.6 - Bố trí số kênh phát trong một khu vực

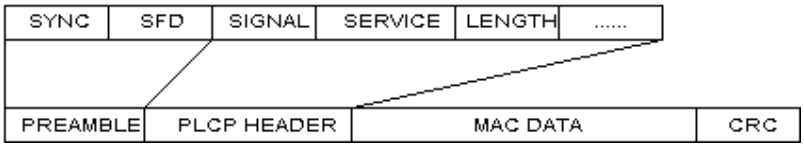


Hình II-4.7 - Khả năng sử dụng lại tần số của phương pháp DSSS

Như vậy trong một vùng đơn tốc độ bit vận chuyển đến có thể lên tới: 11Mbps x 3 = 33Mbps, thay vì 11Mbps như khi chỉ có một kênh truyền được sử dụng trong một khu vực.

4.1.2 Cấu trúc khung PLCP (General PLCP Frame Format)

802.11 PLCP frame bao gồm các thành phần sau:

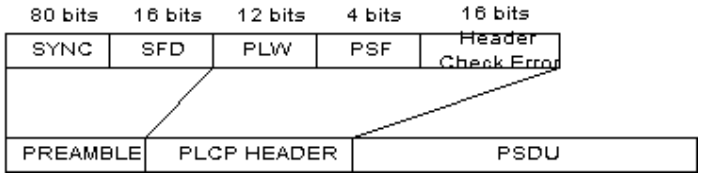


Hình II-4.8 - Dạng chung của PLCP Frame

Preamble (tiêu đề) là thành phần phụ thuộc của PHY. Nó bao gồm hai phần: SYNC (Synchronization - đồng bộ) và SFD (Start Frame Delimiter –ranh giới bắt đầu khung). SYNC là một dãy liên tiếp các số 0 và 1, được lớp vật lý dùng để lựa chọn anten tương ứng, và để đạt được trạng thái giải mã chắc chắn và đồng bộ của bit clock. SFD được sử dụng để xác định sự điều hoà thời gian khung.

PLCP truyền tải ở tốc độ 1 Mbps và bao gồm thông tin logic được lớp PHY sử dụng để giải mã khung. Trường tín hiệu PLCP (PLCP signalling field) chứa thông tin về tốc độ. Trường Service (dịch vụ) sẽ được sử dụng cho các mục đích sau này. Trường Length (độ dài) chỉ ra độ dài dữ liệu MAC hoặc số micro giây cần để truyền dữ liệu MAC (MAC data).

4.1.3 PLCP trong dải phổ dịch tần FHSS

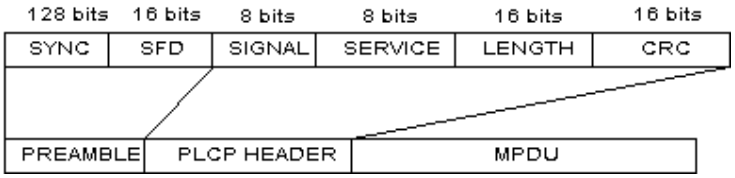


Hình II-4.9 - PLCP trong FHSS

Trường tiêu đề SYNC là một trường 80 bit bao gồm một dãy 0-1 liên tiếp, được truyền bắt đầu từ 0 và kết thúc ở 1. SFD bao gồm dãy 16 bit nhị phân 0000 1100 1011 1101. Bit đầu tiên của SFD cho theo sau bít cuối cùng của dãy SYNC.

PLW (độ dài từ PSDU) chỉ ra số octet trong PSDU. PSF 4 bit (trường tín hiệu PLCP) xác định tốc độ đường truyền, nằm trong khoảng 1 tới 4.5 Mbps, độ lệch 0.5 Mbps. Bit đầu tiên của PSF dùng cho các nhu cầu sau này.

4.1.4 PLCP cho DSSS và HR/DSSS



Hình II-4.10 - PLCP cho DSSS and HR/DSSS

- Trường SYNC:

  Trường Sync bao gồm 128 bit được xáo trộn các bit bằng "1" . Trường này được cung cấp bởi vậy bên nhận có thể thực hiện hoạt động đồng bộ cần thiết. Trạng thái khởi tạo của bộ xáo trộn là [1101100], bit trái nhất xác định giá trị để đưa vào phần tử trễ đầu tiên, bit phải nhất xác định giá trị đưa vào phần tử trễ cuối cùng trong bộ xáo trộn

- SFD được cung cấp để chỉ ra vị trí bắt đầu của tham số phụ thuộc PHY trong PLCP preamble. SFD chiếm 16 bit [1111 0011 1010 0000], bit phải nhất sẽ được truyền đầu tiên đồng bộ.

- Trường SIGNAL

  Trường SIGNAL 8 bit chỉ ra tầng vật lý, sự điều biến sẽ được sử dụng cho việc truyền và nhận PSDU. Tốc độ dữ liệu truyền được cân bằng với giá trị trong trường SIGNAL nhân với 100 Kbit/s. Tầng High Rate PHY cũng hỗ trợ 4 tốc độ bắt buộc bởi các từ 8 bit thiết lập lại tốc độ với đơn vị là 100 Kbit/s và bít ít ý nghĩa nhất được truyền đầu tiên đúng nhịp

  '0X0A' đối với 1 Mbps.

  '0X14' cho 2Mbps

  '0X37' cho 5.5 Mbps

  '0X6E' cho 11 Mbps.

  Trường này được bảo vệ bởi chuỗi kiểm tra frame CCITT CRC-16.

- Trường SERVICE

  Ba bit đã được xác định trong trường SERVICE để hỗ trợ phần mở rộng High Rate. Bit phải nhất (bit 7) được sử dụng để bổ sung trường độ dài. Bit thứ 3 được sử dùng để chỉ ra phương thức điều biến là CCK nếu bằng 0, là PBCC nếu bằng 1. Bit thứ 2 được sử dụng để chỉ ra tần số truyền và các đồng hồ biểu tượng được nhận từ cùng máy tạo dao động. Các đồng hồ khoá này được thiết lập bởi tầng vật lý trên cơ sở cấu hình bổ sung của nó. Trường SERVICE được truyền bit b0 đầu tiên, đúng nhịp, được bảo vệ bởi chuỗi kiểm tra frame CCITT CRC-16.

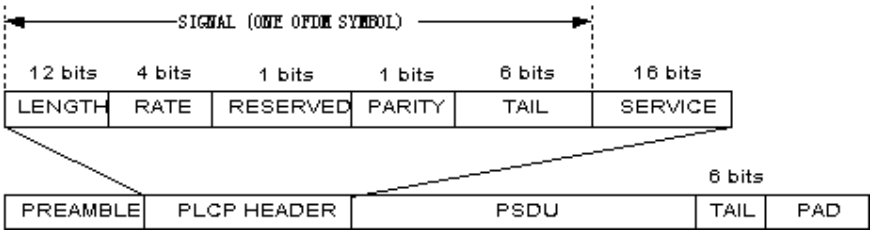
- Trường LENGTH

  Trường PLCP LENGTH được đánh dấu bởi 16 bit số nguyên, chỉ ra số micro giây yêu cầu để truyền PSDU. Giá trị được truyền sẽ được xác định từ tham số LENGTH và DataRate trong TXVECTOR phát ra cùng PHY-TXSTART.request.

- Trường CRC (CCITT CRC-16)

  Các trường SIGNAL, SERVICE và LENGTH được bảo vệ bởi chuỗi kiểm tra frame CCITT CRC-16 (FCS). CCITT CRC-16 FCS là sự bổ sung số dư của 1 được tạo ra bởi phép chia modul 2 của trường PLCP bảo vệ trong đa thức

4.1.5 PLCP trong OFDM



Hình II-4.11 - PLCP trong OFDM

- Tiêu đề PLCP bao gồm 10 ký hiệu (symbol) ngắn và hai ký hiệu dài.
- Trường tín hiệu (SIGNAL) bao gồm độ dài, tốc độ và các trường khác. Trường Rate truyền thông tin về dạng điều biên và tốc độ mã hoá dùng trong phần còn lại của gói tin.
- Bảng giá trị và ý nghĩa của trường RATE

RATE	1011	1111	1010	1110	1001	1101	1000	1100
Data rate	6 Mbps	9 Mbps	12 Mbps	18 Mbps	24 Mbps	36 Mbps	48 Mbps	54 Mbps

4.1.6 Thủ tục truyền PLCP

Các thủ tục truyền đối với High Rate PHY sử dụng long PLCP preamble và header tương tự với sự mô tả trong chuẩn IEEE Std 802.1999 Edition, và không thay đổi thành phần từ khả năng truyền 8.5 Mbit/s và 11 Mbit/s

Tín hiệu PHY-TXSTART.request nguyên thuỷ (TXVECTOR) đượ phát ra bởi MAC để bắt đầu quá trình truyền PPDU. Thêm vào DATARATE và LENGTH, các tham số truyền khác như PREAMBLE_TYPE và MODULATION cũng được thiết lập thông qua PHY-SAP cùng với PHY-TXSTART.request (TXVECTOR) như được miêu tả..

PLCP sẽ phát ra PMD_ANTSEL, PMD_RATE và PMD_TXPWRLVL nguyên thủy để cấu hình tầng vật lý. Sau đó PLCP sẽ phát ra tín hiệu PMD_TXSTART.request và thực thể vật lý sẽ ngay lập tức khởi tạo việc trộn dữ liệu và truyền PLCP preamble trên cơ sở các tham số được chuyển qua trong tín hiệu PHY-TXSTART.request nguyên thuỷ. Thời gian cần thiết cho TX hoạt động trong ramp bao gồm cả trường đồng bộ PLCP. Nếu việc truyền một PLCP preamble hoàn thành, dữ liệu sẽ được trao đổi giữa MAC và PHY bởi một seri tín hiệu PHY-DATA.request (DATA) được phát ra bởi MAC và PHY-DATA.confirm phát ra từ PHY.

4.1.7 Thủ tục nhận PLCP

Thủ tục nhận cho bên nhận được cấu hình để nhận PLCPs tuỳ chọn và bắt buộc, tốc độ dữ liệu, và sự điều biến đã được mô tả. Bên nhận hỗ trợ phần mở rộng High Rate của chuẩn tương ứng 5.5 Mbit/s và 11 Mbit/s hay 1Mbit/s và 2 Mbit/s. Nếu tầng vật lý bổ sung lựa chọn Short Preamble, nó sẽ tìm ra định dạng của cả short và Long Preamble và chỉ ra loại của preamble nhận được trong RXVECTOR. Nếu tầng vật lý bổ sung lựa chọn điều biến PBCC, nó sẽ tìm sự điều biến PBCC hay CCK, như đ• được chỉ ra trong trường SIGNAL, và sẽ thông báo loại điều biến được sử dụng trong RXVECTOR.

Bên nhận sẽ bổ sung thủ tục CCA sẽ được xác định sau. Khi nhận được một PPDU, bên nhận sẽ phân biệt sự khác nhau giữa đinh dạng long và short header bằng giá trị SFD. Bên nhận sẽ giải điều biến long PLCP header sử dụng BPSK ở tốc độ 1 Mbit/s, short PLCP sử dụng QPSK ở tốc độ 2 Mbit/s. Bên nhận sử dụng trường SIGNAL, SERVICE của PLCP header để xác định tốc độ và sự điều biến dữ liệu của PSDU.

Để nhận được dữ liệu, tín hiệu PHY-TXSTART.request sẽ không có hiệu lực, vì vậy thực thể PHY ở trạng thái nhận. Thông qua việc quản lý các trạm bằng PLME, PHY sẽ thiết lập kênh thích hợp và chọn phương thức CCA. Các tham số nhận khác như RSSI (receive signal strength indication), SQ (signal quality), và DATARATE định hướng, có thể được truy nhập thông qua PHY-SAP.

Khi nhận nguồn năng lượng truyền, theo sự lựa chọn phương thức CCA, PMD_ED sẽ có khả năng như RSSI theo ED_THRESHOLD, và/ hoặc PMD_CS sẽ có hiệu lực sau khi khoá mã được thiết lập. Điều kiện để chỉ ra hoạt động của MAC thông qua PHY-CCA.indicate sẽ được phát ra cho năng lượng tìm kiếm và ưu tiên khoá mã để điều chỉnh việc đón nhận PLCP header. Tham số PMD, PMD_SQ và PMD_RSSI được phát ra để update các tham số RSSI và SQ để thông báo với MAC.

Sau khi PHY-CCA.indicate được phát ra, thực thể PHY sẽ bắt đầu tìm kiếm trường SFD. Khi trường SFD được phát hiện, quá trình xử lý CCITT CRC-16 sẽ được khởi tạo và các trường PLCP SIGNAL, SERVICE, và LENGTH sẽ được nhận. CCITT CRC-16 sẽ được xử lý. Nếu CCITT CRC-16 kiểm tra sai, PHY receiver sẽ trở về trạng thái RX IDLE. Khi trạng thái của CCA trở về trạng thái IDLE trong suốt thời gian nhận ưu tiên để hoàn thành quá trình xử lý đầy đủ PLCP, PHY receiver quay lại trạng thái RX IDLE.

Nếu việc nhận PLCP header là thành công (và trường SIGNAL được nhận dạng hoàn toàn và được hỗ trợ), PHY-RXSTART.indicate (RXVECTOR) sẽ được phát ra. RXVECTOR kết hợp với các tham số sau

- Trường SIGNAL
- Trường SERVICE
- Độ dài PSDU theo byte (tính từ trường LENGTH theo đơn vị micro giây và DATARATE theo Mbps)
- RXPREAMBLE_TYPE (là một loại phần tử liệt kê, lấy giá trị SHORTPREAMBLE hay LONGPREAMBLE)
- Thành phần nhận tín hiệu để nhận RSSI và SQ (RX_ANTENNA)

Bit nhận PSDU được kết hợp trong các byte và được đưa tới MAC sử dụng seri trao đổi PHY-DATA.indicate (DATA). Tốc độ và sự thay đổi điều biến được chỉ ra trong trường SIGNAL sẽ được khởi tạo cùng với biểu tượng đầu tiên của PSDU. PHY xử lý việc nhận PSDU. Sau khi nhận bit final của byte PSDU cuối cùng, được chỉ ra trong trường PLCP preamble LENGTH, bên nhận sẽ trở lại trạng thái RX IDLE.

PHY-RXEND.indicate (NoError) sẽ được phát ra. PHY-CCA.indicate (IDLE) được phát ra theo sự thay đổi trong PHYCS (PHY carrier sense) và / hoặc PHYED (PHY energy detection) theo sự lựa chọn phương thức CCA.

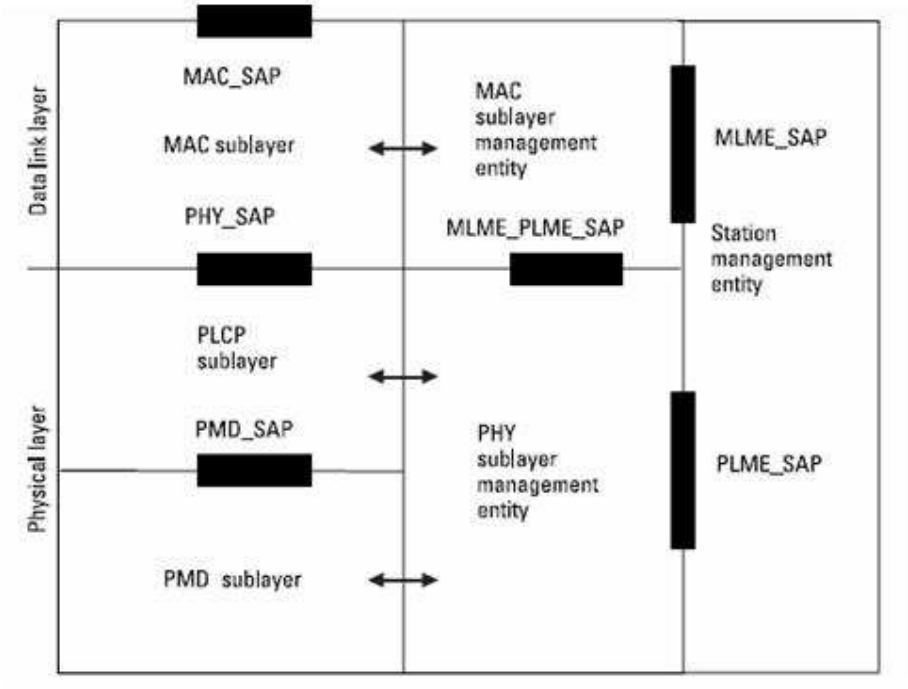
Trong sự kiện mà sự thay đổi trong PHYCS hoặc PHYED có thể làm trạng thái của CCA trở về trạng thái IDLE trước khi hoàn thành việc nhận PSDU, như đã được chỉ ra trong trường PLCP LENGTH, điều kiện lỗi error PHY-RXEND.indicate (CarrierLost) sẽ được thông báo cho MAC. High Rate PHY sẽ đảm bảo CCA đang chỉ ra đường truyền bận trong quá trình truyền PPDU.

Nếu PLCP header thành công, nhưng tốc độ chỉ ra hay sự điều biến trong các trường SIGNAL, SERVICE không tương thích với bên nhận, PHY-RXSTART.indicate sẽ không được phát ra. PHY sẽ phát ra điều kiện lỗi PHY-RXEND.indicate (UnsupportedRate). Nếu PLCP header là không có giá trị, PHY-RXSTART.indicate sẽ không được phát ra, và PHY sẽ phát ra điều kiện lỗi PHY-RXEND.indicate (FormatViolation). Trong cả hai trường hợp, High Rate PHY sẽ đảm bảo CCA chỉ ra đường truyền bận cho việc truyền PSDU. Khoảng thời gian dự định được chỉ ra bằng trường LENGTH (LENGTH x 1)

4.1.8 Tầng PMD (PMD sublayer)

PMD (Physical Medium Dependent sublayer: Tầng vật lý trung gian phụ thuộc) thực hiện việc điều chế (modulation) và mã hóa (encoding).

PMD sublayer cung cấp phương thức và cách thức truyền nhận dữ liệu thông qua WM giữa hai hay nhiều trạm STA, mỗi cái đều sử dụng hệ thống High Rate.Quan hệ giữa hai tầng:



Hình II-4.12 - Tầng PMD

4.2 Tầng kiểm soát truy nhập đường truyền – MAC

Tầng MAC định nghĩa 2 phương pháp truy nhập cơ bản: Distributed Coordination Function và Point Coordination Function

4.2.1 DCF - Distributed Coordination Function

DCF dựa trên cơ chế CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) – Đa truy cập dùng sóng mang có tránh xung đột. Cơ chế đa truy cập dùng sóng mang – CSMA đã rất quen thuộc trong Ethernet, với phương thức Đa truy cập dùng sóng mang có phát hiện xung đột – Carrier Sense Multiple Access with Collision Detection (CSMA/CD).

Phương thức CSMA hoạt động như sau:

- Hoạt động cơ bản của CSMA/CD là "Listen before Talk". Đây là phương pháp mà đôi khi người ta vẫn gọi nôm na là "nghe trước nói sau". Cụ thể là:
    - Khi 1 nút có nhu cầu truyền dữ liệu, nó phải "nghe" xem trên đường truyền có tín hiệu sóng mang hay không. Tức là kiểm tra xem có nút nào khác đang sử dụng đường truyền hay không (Khái niệm sóng mang ở đây có thể hiểu là các dòng điện chứa các tín hiệu sẽ được tầng vật lý chuyển đổi thành các bit dữ liệu).
    - Trong trường hợp đường truyền có sóng mang, thì nút này không được phép truyền, phải chờ - tiếp tục "nghe".

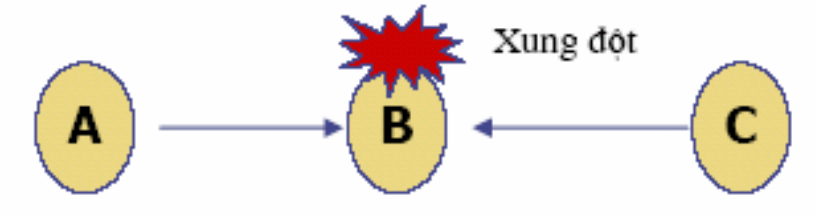
- Nếu không có sóng mang, nút mạng sẽ bắt đầu truyền dữ liệu. Khi đó nó sẽ tạo ra 1 sóng mang trên đường truyền. Và theo quy định của Ethernet, tại 1 thời điểm chỉ có 1 sóng mang trên đường truyền.

- Hiện tượng xung đột xảy ra khi các nút "nghe" đường truyền xong, thấy không có sóng mang, thì cùng truyền tại 1 thời điểm. Vì thế, CSMA được cải tiến bằng cách thêm tính năng phát hiện xung đột (CD – Collision Detection):
  - Nút mạng khi thấy đường truyền rỗi, thì có thể phát hoặc không phát tín hiệu (với xác suất p nào đó).
  - Sau khi bắt đầu truyền, nút truyền vẫn tiếp tục "nghe" thêm 1 khoảng thời gian nữa – Listen while Talk.
  - Trong quá trình nghe này, nếu nút mạng phát hiện ra các sóng mang khác thường (không giống với sóng mang mà mình phát đi, chẳng hạn không giống về cường độ dòng điện…), thì coi như đã có đụng độ xảy ra. Khi đó nút đang truyền sẽ chấm dứt ngay việc phát đi dữ liệu cần truyền, nhưng vẫn tiếp tục phát sóng mang trong 1 khoảng thời gian nữa, để đảm bảo tất cả các nút khác đều nghe được sự đụng độ này.
  - Sau đó nút sẽ tạm nghỉ (Backoff) trong 1 khoảng thời gian ngẫu nhiên(giải thuật random backoff) . Nhờ đó,thời điểm các nút truyền lại sẽ khác nhau.

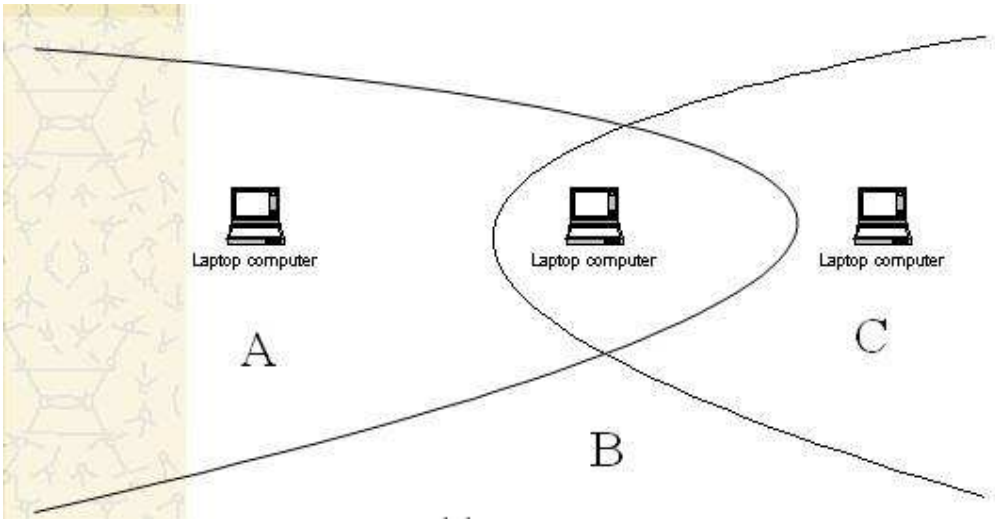
Trong khi cơ chế phát hiện xung đột (Collision Detection) hoạt động rất tốt trong các mạng LAN hữu tuyến, thì đối với mạng LAN vô tuyến, ta không thể sử dụng cơ chế này được. Trong môi trường vô tuyến, ta không thể đảm bảo được tất cả các nút mạng đều "nghe" lẫn nhau được. Đó là vì có hiện tượng "nút ẩn" (Hidden Node).Vả lại các nút mạng trong WLAN không thể thực hiện "Listen while talk" (vừa "nghe", vừa truyền dữ liệu).

Vấn đề nút ẩn như sau:

Đầu cuối ẩn:

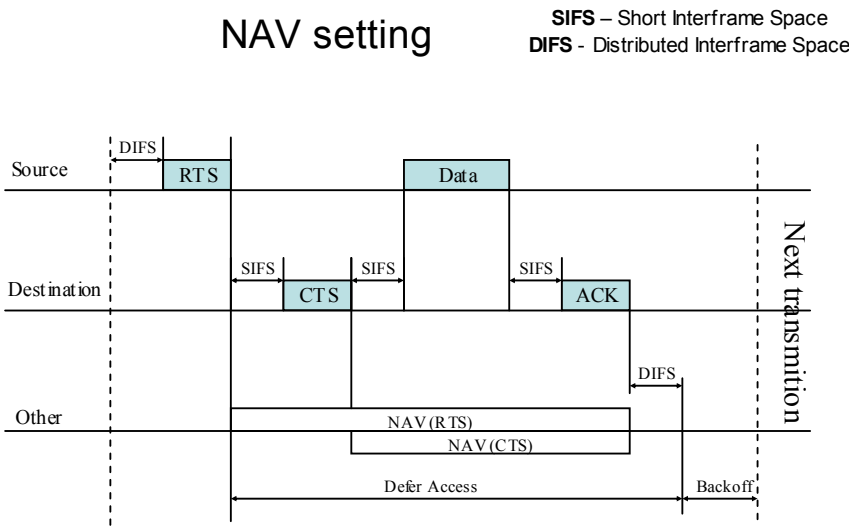


- A trao đổi dữ liệu với B
- C ra nghe kênh truyền
- C không nghe thấy A do C nằm ngoài vùng phủ sóng của A



- C quyết định truyền dữ liệu tới B
- Tại B xảy ra xung đột.

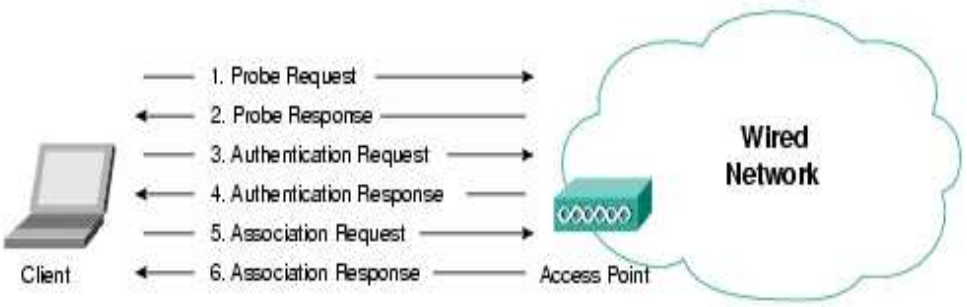
Để giải quyết vấn đề này, người ta sử dụng cơ chế tránh xung đột (CA – Collision Avoidance). Đầu tiên, cũng giống như cơ chế CSMA/CD, nút mạng muốn truyền tin sẽ "nghe" đường truyền. Nếu nó thấy đường truyền đang bận, thì sẽ chờ. Đồng thời nút mạng sẽ tính toán một khoảng thời gian chờ ngẫu nhiên (DIFS). Ngay sau khi thời gian đó trôi qua, nó lại nghe xem liệu có nút mạng nào đang truyền tin hay không. Bằng cách tạo ra thời gian chờ ngẫu nhiên, sẽ hạn chế được hiện tượng các nút mạng muốn truyền tin sẽ truyền tại cùng một thời điểm (tránh xung đột). Ngược lại, khi đường truyền rỗi, nút mạng sẽ được phép truyền tin. Tuy nhiên nút mạng không truyền dữ liệu ngay, mà nó sẽ phát sẽ phát đi một gói tin báo gửi là RTS - Request To Send (chi tiết về cấu trúc của gói tin này sẽ được trình bày sau). Nút nhận nếu cũng thấy đường truyền đang rỗi, thì sẽ phản hồi lại bằng gói tin cho phép gửi - CTS (Clear To Send). Khi đó cặp RTS/CTS được coi là một sóng mang ảo (Virtual Carrier Sense). Sở dĩ gọi như vậy vì, sóng mang ảo này cũng có vai trò như sóng mang trong cơ chế CSMA/CD. Tức là, khi các nút khác ra "nghe" đường truyền mà phát hiện ra có sóng mang ảo, thì chúng sẽ coi đường truyền đang bận. Sóng mang ảo còn được gọi là vector thiết lập liên kết NAV – Network Allocation Vector. Sau khi thiết lập RTS/CTS, nút mạng bắt đầu truyền dữ liệu. Và trong quá trình truyền này có sử dụng cơ chế báo nhận (ACK). Tức là, nút nhận sau khi nhận được dữ liệu sẽ gửi lại thông báo nhận – ACK (Acknowledgement). Toàn bộ tiến trình trên gọi là 4-way handshake.



Hình II-4.13 - Cơ chế 4-way handshake

4.2.2 PCF – Point Coordination Function

4.2.3 Phân tích các hoạt động cơ bản



Hình II-4.14 - Các hoạt động cơ bản

4.2.3.1 Scanning

Khi ta cài đặt (install), cấu hình hay khởi động một thiết bị khách (WLAN client device), thì các client này sẽ tự động kiểm tra xem nó có ở trong vùng phủ sóng của một AP nào đó không. Đồng thời xác định xem nó có thể tham gia (associate) vào mạng WLAN này hay không. Quá trình này gọi là scanning. Như vậy scanning là tiến trình xảy ra trước mọi tiến trình khác. Đó là cách mà các client tìm ra mạng WLAN.

Để hỗ trợ các client xác định các AP, có 1 số công cụ như là: SSID – Service Set Identifier và frame "hoa tiêu" hay còn gọi là frame dẫn đường (beacons). Đồng thời, có 2 hình thức Scanning: bị động và chủ động.

SSID

SSID với mỗi mạng thường là duy nhất, phân biệt chữ hoa – thường (case sentitive). Thường gồm từ 2 tới 32 ký tự. Và được sử dụng làm tên mạng (network name).

Giá trị của SSID được gửi đi trong các frame dẫn đường (beacons), probe request, probe response…

Frame dẫn đường

Tên đầy đủ là Beacon Management Frame: là các frame được phát từ AP tới các trạm làm việc - station (nếu ở chế độ cơ sở), hoặc giữa các trạm với nhau (nếu ở chế độ độc lập – ad hoc mode).

Frame định hướng chứa các thông tin về đồng bộ thời gian, các tham số trải phổ, SSID, tốc độ cho phép…. Cụ thể:

- Thông tin đồng bộ thời gian
    - Khi client nhận được các frame định hướng, nó sẽ thay đổi đồng hồ (clock) của mình sao cho tương ứng (reflect) với đồng hồ của AP. Khi đó 2 đồng hồ được gọi là đồng bộ
    - Tác dụng: đảm bảo tất cả các công việc phân biệt bởi thời gian (time sensitive funtions) như là nhảy tần (hopping) trong FHSS được thực hiện mà không bị lỗi.
- Các thông số trải phổ
    - Các frame dẫn đường chứa thông tin về kỹ thuật trải phổ mà hệ thống đang sử dụng. Các thông tin này được thể hiện dưới dạng tập các tham số FH hoặc DS.
    - VD:

      Với 1 hệ thống FH là các tham số về hop, dwell time và hop sequence - các thông số về thời gian nhảy và ngừng (ý nghĩa các tham số này xin xem ở phần phụ lục).

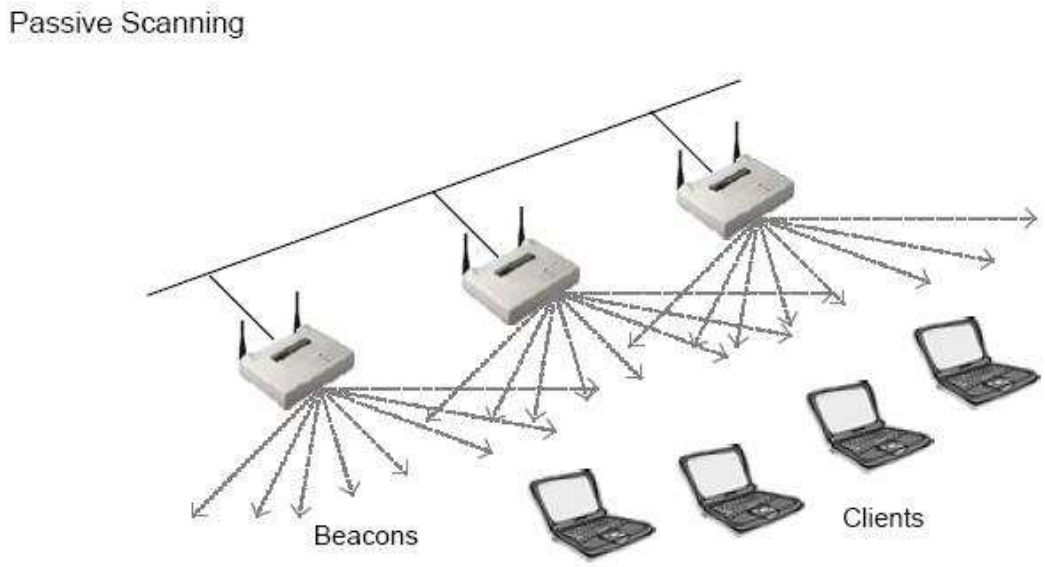
      Với hệ thống DS: frame dẫn đường sẽ chứa thông tin về các kênh.
- SSID
    - Station sau khi nhận được các frame dẫn đường, sẽ căn cứ vào các frame này để xác định SSID của mạng mà nó muốn kết nối. Sau khi tìm thấy thông tin SSID (network name), station sẽ tìm địa chỉ MAC của nơi phát ra các frame dẫn đường. và gửi 1 yêu cầu cần xác thực (authentication request), để chờ được kết nối(associate) vào mạng.
    - Nếu một trạm được đặt chế độ chấp nhận bất kỳ SSID nào, thì sau đó nó sẽ cố gắng tham gia vào mạng thông qua AP đầu tiên gửi frame dẫn đường, hoặc AP nào có tín hiệu mạnh nhất (trong trường hợp có nhiều AP).
- Supported rate (tốc độ cho phép):

  Trong các beacons còn chứa thông tin về tốc độ cho phép của AP . VD: các thiết bị chuẩn 802.11b hỗ trợ tốc độ 1 Mbps, 2 Mbps, 5.5 Mbps và 11 Mbps.

Tiếp theo ta sẽ tìm hiểu 2 dạng Scanning

Passive Scanning (Quét bị động)

Là quá trình AP gửi các frame dẫn đường tới các station (nếu ở mô hình cơ sở), hoặc các station gửi frame dẫn đường cho nhau (nếu ở mô hình độc lập – ad hoc mode). Các station sẽ scanning để xác định các đặc tính của phía phát dựa vào các frame dẫn đường này.



Hình II-4.15 - Passive Scanning

Station nào muốn tìm mạng sẽ "lắng nghe" các frame dẫn đường cho đến khi phát hiện ra một frame dẫn đường có chứa định danh - SSID của mạng mà nó muốn kết nối. Tiếp theo, station sẽ kết nối vào mạng đó thông qua AP nào đã gửi frame dẫn đường cho nó. (Quá trình kết nối vào mạng tiếp theo như thế nào sẽ được trình bày sau).

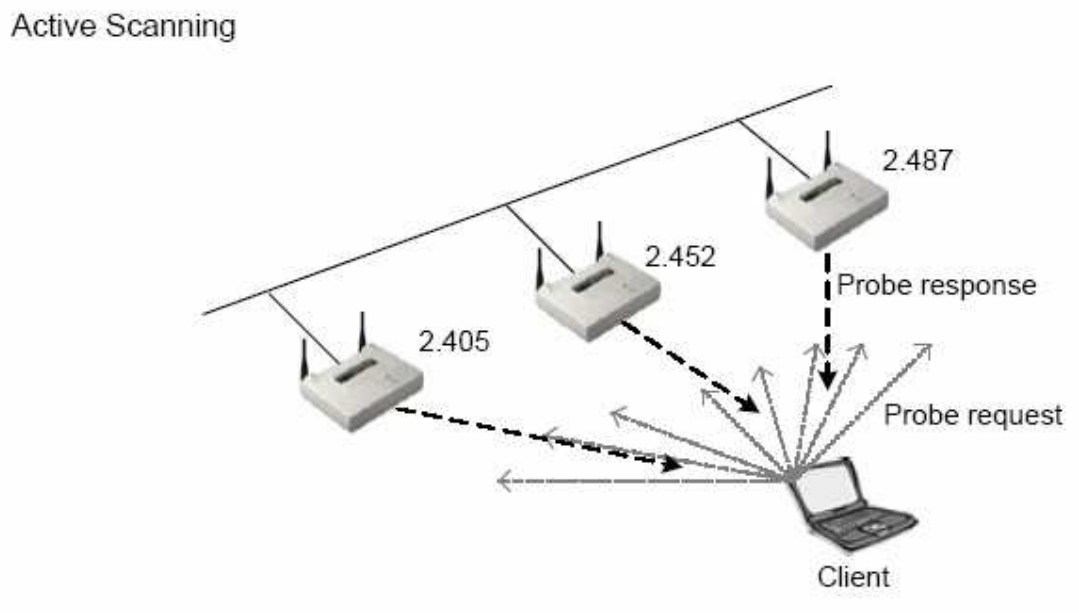
Trong trường hợp có nhiều AP, thì sẽ có nhiều frame dẫn đường của các mạng khác nhau. Khi đó, station sẽ kết nối vào mạng thông qua AP có tín hiệu mạnh nhất (thông tin về độ mạnh yếu của tín hiệu cũng có trong các frame dẫn đường mà AP phát đi) hay AP có tỷ lệ bit lỗi ít nhất (lowest bit error rate).

Sau khi đã kết nối vào mạng rồi, máy trạm vẫn duy trì danh sách các AP có thể dùng để kết nối và đặc tính kèm theo (số lượng kênh, độ mạnh yếu của tín hiệu, SSID…), để khi cần nó có thể kết nối lại với AP. Ngoài ra station có thể dịch chuyển (roaming) từ AP này sang AP khác (VD trong trường hợp tín hiệu của AP mà nó đang kết nối đột nhiên bị yếu đi). Khi đó, nhờ danh sách các AP sẵn có, máy trạm sẽ nhanh chóng xác định được AP cần kết nối.

Active Scanning (quétchủ động)

Máy trạm sẽ gửi các frame thăm dò (Probe request frame). Trong các frame này có chứa định danh của một mạng cụ thể mà trạm muốn kết nối tới, hoặc một định danh mạng quảng bá (broadcast SSID).

Trong trường hợp frame thăm dò chứa SSID của 1 mạng cụ thể, thì chỉ AP nào phục vụ cho mạng này mới phản hồi lại. Còn nếu là broadcast SSID, thì bất kỳ AP nào nhận được frame thăm dò từ máy trạm, cũng có thể phản hồi lại.



Hình II-4.16 - Active Scanning

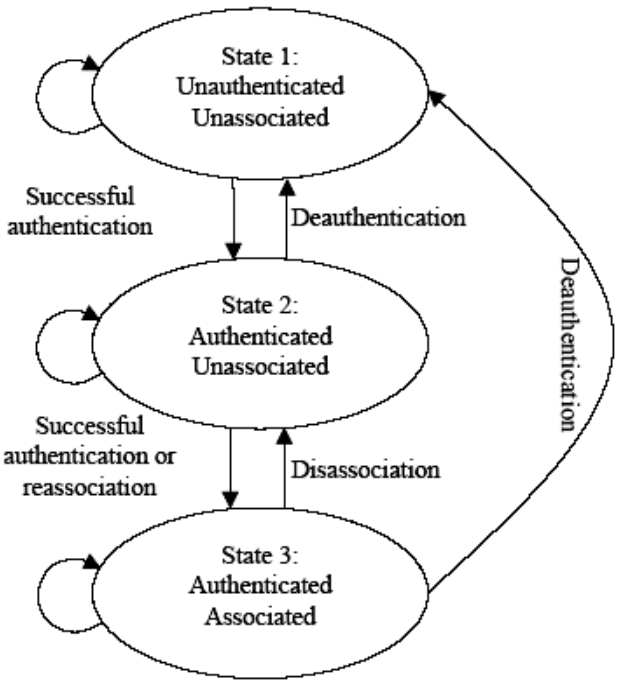
Một khi máy trạm đã xác định được AP thích hợp, nó sẽ thực hiện tiếp quá trình xác thực và liên kết.

4.2.3.2 Xác thực và Liên kết(Authentication & Association)

Quá trình này có ba trạng thái phân biệt:

- Không chứng thực và không liên kết (Unauthenticated and unassociated)
- Chứng thực và không liên kết (Authenticated and unassociated)
- Chứng thực và liên kết (Authenticated and associated)

Ba trạng thái này diễn ra theo sơ đồ sau:



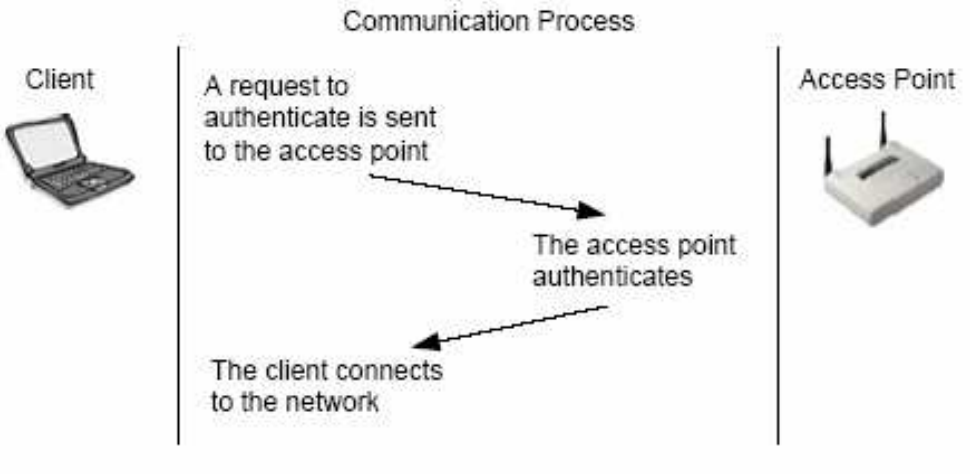
Hình II-4.17 - Các phương pháp chứng thực cơ bản

Các phương pháp chứng thực cơ bản

Chứng thực hệ thống mở

Quá trình này được thực hiện một cách đơn giản theo hai bước sau:

- Máy client gửi một yêu cầu chứng thực tới AP
- AP chứng thực máy khách và gửi một trả lời xác thực client được liên kết



Hình II-4.18 - Chứng thực hệ thống mở

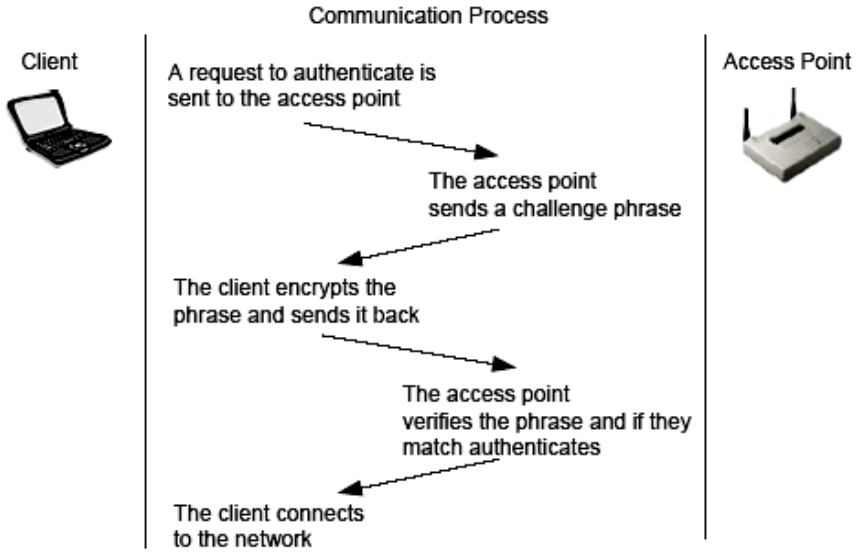
Phương pháp này được cài đặt mặc định trong các thiết bị WLAN 802.11. Nhờ đó, một trạm có thể liên kết với bất cứ một AP nào sử dụng phương pháp chứng thực hệ thống mở, khi nó có SSID đúng. SSID đó phải phù hợp trên cả AP và client, trước khi client đó hoàn thành quá trình chứng thực. Trong phương pháp này thì WEP chỉ được sử dụng để mã hóa dữ liệu, nếu có.

Chứng thực khóa chia sẻ

Phương pháp này bắt buộc phải dùng WEP.

Một quá trình chứng thực khóa chia sẻ xảy ra theo các bước sau:

- Clien gửi yêu cầu liên kết tới AP (bước này giống như chứng thực hệ thống mở).
- AP sẽ gửi một đoạn văn bản ngẫu nhiên tới Client, văn bản này chưa được mã hóa, và yêu cầu Client dùng chìa khóa WEP của nó để mã hóa.
- Clien mã hóa văn bản với chìa khóa WEP của nó và gửi văn bản đã được mã hóa đó đến AP.
- AP sẽ thử giải mã văn bản đó, để xác định xem chìa khóa WEP của Client có hợp lệ không. Nếu có thì nó gửi một phản hồi cho phép, còn nếu không, thì nó trả lời bằng một thông báo không cho phép client đó liên kết.



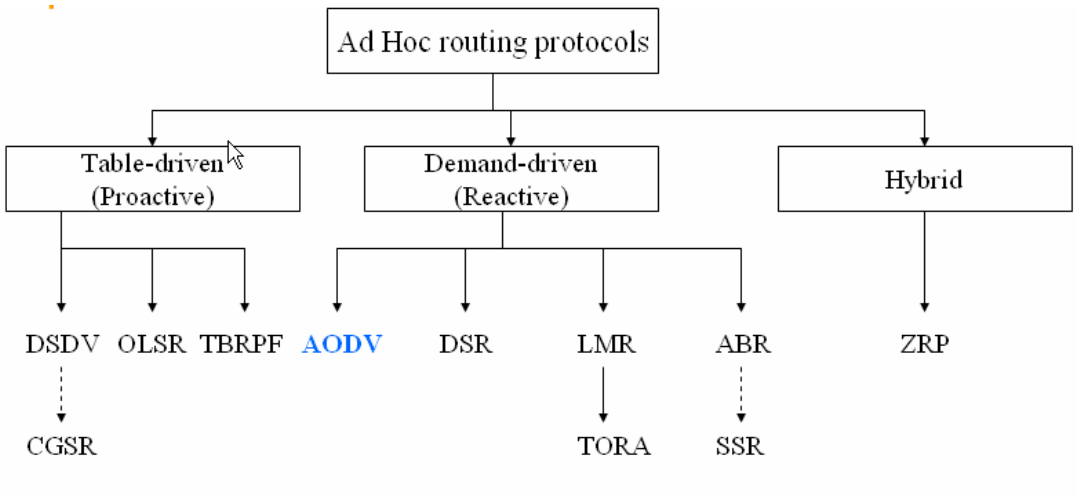
Hình II-4.19 - Chứng thực khoá chia xẻ

Nhìn qua thì phương pháp này có vẻ an toàn hơn phương pháp chứng thực hệ thống mở (vì có nhiều bước hơn). Tuy nhiên, nếu xem xét kỹ thì trong phương pháp này, chìa khóa WEP được dùng cho hai mục đích: để chứng thực và để mã hóa dữ liệu. Đây chính là kẽ hở để hacker có cơ hội thâm nhập mạng. Hacker sẽ thu cả hai bản tin: văn bản chưa mã hóa do AP gửi và văn bản đã mã hóa, do Client gửi. Và từ hai thông tin đó hacker có thể giải mã ra được chìa khóa WEP.

4.3 Tầng mạng và các giao thức dẫn đường trong WLAN

Như trên đã nói, WLAN gồm 2 cấu hình mạng cơ bản: BSS và Adhoc. Đối với BSS, ta chủ yếu nghiên cứu giao thức Mobile IP. Còn riêng với Adhoc thì có nhiều giao thức dẫn đường. Rõ ràng, những đặc điểm chủ yếu của các mạng kiểu ad-hoc (khả năng di động cao, không có quyền quản lý trung tâm, tài nguyên hữu hạn...) bắt buộc trong mô hình mạng phải có chức năng tìm đường. Một giao thức tìm đường thích hợp với MANET sẽ không chỉ có khả năng phản ứng nhanh với các mạng có cấu trúc

topo thay đổi (cấu trúc động) mà nó còn phải có khả năng duy trì chất lượng quá trình truyền tin trong điều kiện tài nguyên giới hạn (băng thông, pin, ...). Các giao thức kinh điển như link-state hay distance-vector có thể đáp ứng được các yêu cầu này tuy nhiên nó rất tốn tài nguyên và thời gian, đặc biệt là trong trường hợp mạng có tính di động cao.



Hình II-4.20 - Các giao thức tìm đường trong mạng Ad-hoc

4.3.1 Các giao thức tìm đường trong mạng Ad-hoc

4.3.1.1 Những yêu cầu thực tế

Khi xây dựng giao thức chọn đường cho mạng ad-hoc chúng ta cần phải chú ý đến những đặc điểm quan trọng sau đây:

Các nút mạng trong MANET được kết nối với nhau qua các liên kết không dây với độ rộng băng thông hạn chế. Do đó giao thức chọn đường thích hợp với MANET phải có khả năng hoạt động với điều kiện băng thông có hạn.

Các nút mạng trong MANET thực tế chính là các thiết bị di động như các máy PDA, laptop. Những thiết bị này có nguồn tài nguyên hữu hạn. Những nguồn tài nguyên mà cụ thể là dung lượng bộ nhớ, năng lượng pin cần phải được sử dụng một cách thông minh.

Các nút mạng trong MANET thường có khả năng di động cao. Điều này đồng nghĩa với việc topo của mạng ad-hoc thường xuyên thay đổi. Các giao thức tìm đường thích hợp với ad-hoc cần phải có khả năng tìm nhanh các nút mạng thay thế khi cấu trúc mạng có biến động. Khả năng phản ứng nhanh (tốc độ hội tụ nhanh - rapid convergence) là một mục tiêu quan trọng đối với một giao thức tìm đường cho mạng ad-hoc.

4.3.1.2 Các giao thức kinh điển

Hai giao thức chủ yếu là link-state và distance-vector thường được sử dụng trong các mạng chuyển mạch gói (packet-switched network). Cả hai giao thức này đều cho phép một nút mạng có khả năng quyết định bước truyền (hop) tiếp theo dựa vào "đường

dẫn ngắn nhất" (shortest path) để tới được nút đích. Đường dẫn ngắn nhất được tính toán dựa trên một ước lượng cụ thể, thường là khoảng cách giữa số các bước truyền.

Giao thức link-state

Trong giao thức link-state, mỗi nút mạng sẽ tập hợp trạng thái (ước lượng để tính đường dẫn ngắn nhất) của tất cả các liên kết đi ra (các liên kết hướng tới các nút lân cận của nó) và lan truyền tập hợp này tới tất cả các nút khác trong mạng nhờ vào việc truyền gói tin trạng thái liên kết (LSP - Link State Packet). Điều này được thực hiện theo chu kỳ hoặc bất cứ khi nào có sự thay đổi ảnh hưởng đến một trong các liên kết trên. Ngoài ra mỗi gói tin LSP sẽ được gán thêm một số thứ tự nhằm xác định độ cập nhật của nó (các gói tin có số thứ tự lớn hơn thì mới hơn).

Khi một nút mạng nhận được các gói tin LSP, nó sẽ cập nhật lại tất cả các trạng thái liên kết (phù hợp với topo mạng mới). Dựa vào những thông tin này, nút mạng sẽ thực hiện thuật toán tìm đường ngắn nhất (thường là thuật toán Dijkstra) và quyết định bước truyền tối ưu tiếp theo cho mỗi nút đích trong mạng. Chỉ có bước truyền tiếp theo và chi phí tìm đường tương ứng với mỗi nút mạng mới được lưu trong bảng định tuyến (routing table).

Phải chú ý rằng các nút mạng không cập nhật lại bảng định tuyến cùng một lúc (do độ trễ khi truyền gói tin LSP) do đó điều này có thể dẫn đến trường hợp đường dẫn ngắn nhất bị quay vòng. Tuy nhiên sự kiện này sẽ mất đi khi tất cả các nút trong mạng đều nhận được gói tin LSP.

Giao thức distance-vector

Thuật toán vector khoảng cách (distance vector algorithm) hay còn được gọi là thuật toán Bellman-Ford dựa trên một phương pháp tiếp cận khác. Mỗi nút mạng sẽ chỉ thông báo phần còn lại của bảng định tuyến của nó cho các nút mạng liên kết trực tiếp với nó. Khi một nút nhận được thông tin từ các nút lân cận trực tiếp, nó sẽ cập nhật lại bảng định tuyến dựa trên bảng định tuyến của các nút lân cận. Trên cơ sở đó, đối với mỗi nút đích, một nút mạng sẽ "nhìn" vào khoảng cách của các nút lân cận của nó (khoảng cách của một nút lân cận là khoảng cách từ nút lân cận đó đến một nút đích) để xác định khoảng cách nhỏ nhất. Như vậy trong giao thức này, mỗi nút mạng sẽ chỉ lưu trữ khoảng cách của tất cả các nút lân cận trực tiếp và có khả năng thay đổi nó khi có biến động.

Nhận xét

Cả hai thuật toán trên đều có chứng minh được tính hiệu quả trên những mạng không có tính di động cao. Tuy nhiên cả hai đều gặp vấn đề khi áp dụng cho những mạng có cấu trúc thay đổi thường xuyên. Quá trình tìm đường không chỉ tốn nhiều thời gian mà còn sử dụng rất lãng phí các tài nguyên (tìm đường đến tất cả các nút mạng nhưng hầu hết trong số này sẽ không được sử dụng).

4.3.2 Các giao thức mở rộng cho MANET

Cùng với sự phát triển nhanh của mạng ad-hoc, một tập lớn các giao thức tìm đường dựa trên địa chỉ IP đã được đề xuất. Chưa có một chuẩn nào được công nhận một cách chính thức, nhưng cũng có một số giao thức hứa hẹn đang được tập trung nghiên cứu. Có thể kể ra hai giao thức chính là AODV (Ad-hoc On Demand Distance Vector) và DSDV (Destination Sequenced Distance Vector).

4.3.2.1 Giao thức DSDV

Ta có thể dễ dàng nhận ra giao thức DSDV là một giao thức dựa trên thuật toán vector khoảng cách.

DSDV có hai sự thay đổi cơ bản nhằm làm cho giao thức phù hợp hơn với mô hình mạng ad-hoc di động. Sự thay đổi đầu tiên là khắc phục vấn đề "đếm tới vô cùng" (counting to infinity) và "hình thành vòng lặp". Ý tưởng này thực tế rất đơn giản, chỉ cần gán cho mỗi đường đi tìm được một số thứ tự đánh dấu độ cập nhật của nó. Trên cơ sở đó một đường đi có số thứ tự lớn hơn sẽ là một đường đi mới hơn tức là nó có khả năng chính xác hơn đối với cấu trúc mạng hiện tại. Ngoài ra các nút đích cũng sinh ra các số thứ tự cho riêng nó. Các số này được duy trì tại mỗi nút và tăng lên mỗi khi có sự thay đổi trên đường đi. Số thứ tự và chi phí đường đi được gán trong các message cập nhật. Do đó khi một nút nhận được một message cập nhật, nó sẽ so sánh số thứ tự nhận được với số thứ tự của nó lưu trong bảng định tuyến. Chỉ có những message cập nhật có số thứ tự lớn hơn mới có tác dụng. Cũng có trường hợp hai số thứ tự này bằng nhau, khi đó đường đi nào có chi phí nhỏ hơn sẽ được cập nhật lại.

Thay đổi thứ hai được thêm vào thuật toán ban đầu nhằm mục đích làm giảm sự quá tải sinh ra bởi hoạt động của giao thức. Nó bao gồm hai kiểu message cập nhật: full dump và incremental. Gói tin full dump được sử dụng để thông báo thông tin về tất cả các đường đi có giá trị sử dụng trong khi đó một gói tin incremental có kích thước nhỏ hơn chỉ được sử dụng để thông báo các thay đổi giữa hai gói tin full dump.

DSDV hoạt động ổn định hơn thuật toán Bellman-Ford gốc. Tuy nhiên do nó vẫn còn dựa vào các việc truyền nhận các message theo chu kỳ để duy trì thông tin tìm đường nên giao thức này vẫn cần có sự điều chỉnh để giảm thiểu khối lượng thông tin lan truyền.

4.3.2.2 Giao thức AODV

Là sự phát triển của DSDV, AODV làm giảm được tổng số các thông tin điều khiển được truyền nhận trong mạng bằng cách cực tiểu hoá số đường đi cần thiết. Thay vì xây dựng đường đi cho toàn bộ các nút đích có thể có trong mạng, mỗi nút mạng sẽ chỉ tạo và duy trì những đường đi mà nó thực sự cần. Khi cần một đường đi, một nút mạng sẽ khởi tạo một yêu cầu để định vị các nút tiếp theo. Nói một cách khác, khi một đường đi không được sử dụng thường xuyên nó sẽ bị xóa khỏi bảng định tuyến. Phương pháp này còn được gọi là phương pháp tìm đường source-initiated on-demand (tạo nguồn khi có yêu cầu) trái ngược với các phương pháp tìm đường cũ dựa hoàn toàn vào bảng định tuyến (table-driven). Cách tiếp cận của AODV còn được gọi là reactive (phản ứng lại) thay cho cách tiếp cận proactive (thực hiện trước).

Ngoài ra còn một thay đổi khác liên quan đến việc duy trì các đường đi. Thực tế, trong trường hợp kết nối không thực hiện được, các nút mạng sẽ lập tức thông tin ngược lại cho tất cả các nút thực sự chịu ảnh hưởng. Các cập nhật theo định kỳ như các gói tin full dump của giao thức DSDV hoàn toàn bị loại bỏ. Tất nhiên, AODV vẫn sử dụng một tập các số thứ tự tương tự như trong DSDV để đảm bảo tránh hiện tượng vòng lặp trên các đường đi.

4.3.3 Mô tả chi tiết giao thức AODV

Như đã trình bầy trong phần 4.3.2.2, giao thức AODV sử dụng một cách tiếp cận hoàn toàn mới so với các phương pháp truyền thống để xây dựng các đường đi trong mạng. Khi một nút mạng muốn gửi một gói tin tới một nút đích nào đó, nó sẽ khởi tạo tiến trình xử lý discovery) để định vị nút đích. Nếu không có một đường đi nào được tìm thấy trong một khoảng thời gian xác định, nút khởi tạo sẽ cho rằng không tồn tại đường đi tới nút đích. Tiến trình xử lý discovery sẽ kết thúc đồng thời các gói tin tương ứng sẽ bị huỷ bỏ. Ngược lại nếu nút khởi tạo tìm được một đường đi phù hợp, nó sẽ cập nhật đường đi này vào bảng định tuyến của nó như một đầu vào (entry) tương ứng với nút đích.

Khi một đầu vài mới được tạo ra, tiến trình xử lý maintenance cũng đồng thời được kích hoạt để giám sát tình trạng của đường đi vừa được tạo ra - nếu sau một khoảng thời gian đủ lớn mà đường đi không được sử dụng, nút mạng sẽ xoá đường đi này ra khỏi bảng định tuyến. Nếu có lỗi xuất hiện trên một đường đi có trạng thái tích cực, nút mạng sẽ lập tức thông báo ngược cho các bước truyền trước đó bằng một kiểu gói tin điều khiển cụ thể. Trong trường hợp nhận được các gói tin thông báo trên, các nút mạng chịu ảnh hưởng sẽ khởi động lại tiến trình discovery để tìm một đường đi thay thế nếu cần thiết.

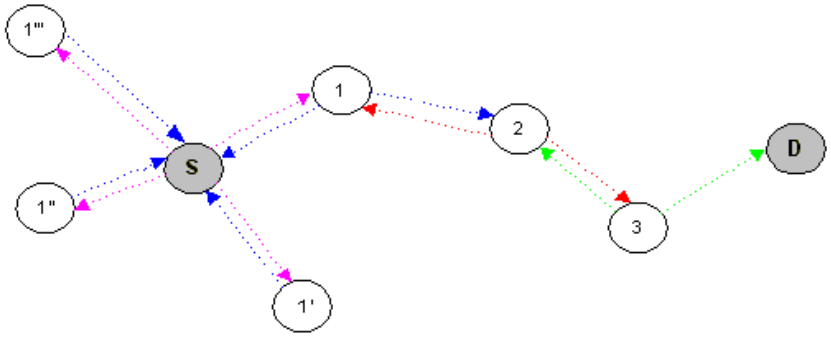
AODV quản lý các thông tin về đường đi theo kiểu phân tán. Điều này có nghĩa là mỗi nút trên đường đi sẽ có một thành phần trong bảng định tuyến tương ứng với nút đích của đường đi đó. Cách quản lý này hoàn toàn trái ngược với phương pháp source routing (tìm đường từ nút nguồn) trong đó chỉ có nút nguồn mới biết đường đi đầy đủ tới nút gốc. AODV cũng cho phép mỗi nút chỉ duy trì một và chỉ một đường đi ứng với mỗi nút đích. Tuy nhiên có một số giao thức tìm đường khác cho phép tìm nhiều đường ứng với nút đích. Trong trường hợp, đường đi ban đầu bị lỗi, đường đi thay thế sẽ được sử dụng.

Các thành phần của bảng định tuyến trong giao thức AODV được định dạng bao gồm các trường <địa chỉ nút đích, địa chỉ bước truyền tiếp theo, số thứ tự, khoảng cách, danh sách các nút trước đó (precursor), ngày hết hạn>. Trường số thứ tự được sử dụng để ngăn chặn sự hình thành các vòng lặp và thể hiện mức độ cập nhật của các đường đi. Khoảng cách là số bước truyền. Khi một đường đi mất hiệu lực, số thứ tự của nó sẽ được tăng lên một và khoảng cách sẽ được đặt là vô cùng. Danh sách precursor chứa tập hợp các nút lân cận, sử dụng thành phần này để chuyển tiếp các gói dữ liệu. Ngày hết hạn được sử dụng để xác định thời gian tồn tại tối đa của thành phần, sau đó nó sẽ bị xoá bỏ khỏi bảng định tuyến. Tất nhiên giá trị của trường này sẽ được mở rộng mỗi lần thành phần này được sử dụng.

4.3.3.1 Tiến trình Discovery

Tiến trình Discovery là một phương pháp kỹ thuật cho phép từng nút nguồn trong một mạng MANET có thể định vị (lấy được địa chỉ IP) một nút đích. Tất nhiên, một nút khởi động tiến trình Discovery chỉ khi nút đích chưa từng được định vị trước đó hoặc không có thành phần nào trong bảng định tuyến tương ứng với nút đích. Nếu đã có một thành phần tồn tại trong bảng định tuyến, gói tin sẽ lập tức được chuyển đi và bước thực thiện tiến trình Discovery được bỏ qua.

Khi một nút khởi động cho tiến trình Discovery, nó phải gửi đi một gói tin yêu cầu tìm đường (route request) (RREQ) tới tất cả các nút lân cận. Gói tin này được lan truyền tới tất cả các nút khác trong mạng cho đến khi một đường đi được xác định.



Hình II-4.21 - Quá trình lan truyền của gói tin RREQ

Tạo Route Request

Nếu một nút nguồn A muốn tìm một nút đích B trong mạng MANET, A sẽ thông báo cho tất cả các nút khác trong mạng biết nó đang tìm kiếm nút B bằng cách gửi đi gói tin RREQ tới tất cả các nút lân cận. RREQ chứa địa chỉ IP và số thứ tự của cả nguồn và đích. Số thứ tự đích tham chiếu đến số thứ tự của đường đi cuối cùng tới nút B mà nút nguồn A biết. Nếu nút nguồn A không dò được một số thứ tự nào của đường đi tới nút B thì số thứ tự đích được đặt mặc định bằng 0.

Mỗi nút trong mạng khi nhận được gói tin RREQ sẽ lập tức lan truyền tiếp tới tất cả các nút lân cận của nó cho đến khi gói tin RREQ tới được nút B hoặc một một nút nào đó biết một đường đi đầy đủ và cập nhật tới nút B (dựa vào số thứ tự đích trong gói tin RREQ).

Gói tin RREQ còn chứa hai trường khác là time to live (TTL) và broadcast ID. Trường TTL cho phép tiến trình Discovery điều khiển mức độ lan truyền của gói tin RREQ trong mạng. Lấy ví dụ như một gói tin RREQ có trường TTL được đặt bằng 2 sẽ thực hiện nhiều nhất 2 bước truyền tính từ nút gốc. Khi gói tin RREQ truyền đi, nút nguồn sẽ đặt giá trị cho trường TTL và chuyển vào trạng thái chờ. Khoảng thời gian chờ của nút gốc tương ứng tỉ lệ với giá trị của trường TTL trong gói tin RREQ. Nếu không có gì thay đổi, một đường đi được tìm thấy trước khi thời gian chờ kết thúc thì tiến trình Discovery kết thúc thành công. Ngược lại, nếu trong khoảng thời gian chờ của nút nguồn nó không nhận được một gói tin trả lời nào, nút nguồn sẽ gửi lại một gói tin RREQ mới và tiếp tục chờ đợi. Tất nhiên gói tin mới sẽ có giá trị của trường TTL lớn hơn và thời gian chờ đợi cũng kéo dài hơn vì thế gói tin RREQ sẽ lan truyền đến được nhiều nút mạng hơn. Nếu vẫn không nhận được trả lời, nút nguồn sẽ gửi tiếp một gói tin RREQ với giá trị tối đa của trường TTL. Sau lần này, tiến trình Discovery sẽ bị huỷ bỏ. Kỹ thuật này còn được gọi là kỹ thuật mở rộng dần vòng tìm kiếm.

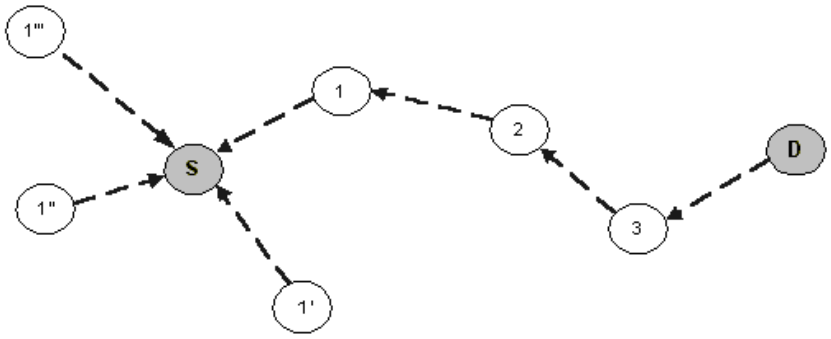
Ngoài ra mỗi gói tin RREQ còn được gán một số thứ tự được gọi là Broadcast ID. Trường này được dùng cho các nút mạng phân biệt các gói tin xuất phát từ cùng một nút và nó được tăng lên sau mỗi lần truyền đi. Một cặp <địa chỉ IP nguồn, Broadcast ID> là duy nhất cho mỗi gói tin RREQ và các gói tin có giá trị của trường Broadcast ID lớn hơn thì mới hơn. Khi một nút nhận được một gói tin RREQ nó sẽ ghi nhớ giá trị của trường Broadcast ID. Sau đó nếu nút đó tiếp tục nhận được các gói tin RREQ từ cùng một nút nguồn thì chỉ những gói tin có giá trị Broadcast ID lớn hơn giá trị mà nó đã ghi nhớ mới được tiến hành xử lý. Các gói tin có giá trị nhỏ hơn sẽ bị huỷ bỏ.

Trong mỗi gói tin RREQ còn có trường hop count được dùng để ghi số bước truyền mà gói tin đã được truyền qua.

Chuyển tiếp Route Request

Giả sử trường hợp một nút I nhận được một gói tin RREQ xuất phát từ nút nguồn S, yêu cầu tìm đường tới nút đích D. Đầu tiên nút I sẽ kiểm tra giá trị của trường Broadcast ID và địa chỉ của nút nguồn. Nếu giá trị của trường Broadcast ID nhỏ hơn hoặc bằng giá trị mà nút đã ghi nhớ trước đó, gói tin RREQ sẽ bị huỷ bỏ. Trong trường hợp ngược lại, nút I sẽ xử lý gói tin RREQ. Trước hết nó sẽ tạo hoặc cập nhật lại đường đi ngược tới nút S (hình dưới mô tả tất cả các đường đi ngược được tạo ra khi gói tin RREQ lan truyền trong mạng). Đường đi này được sử dụng để chuyển thông báo trả lời - route reply trở lại nút S trong trường hợp đường đi tới nút D được tìm thấy.

Sau khi một đường đi ngược được tạo ra, nút I sẽ kiểm tra xem nó có biết một đường đi đầy đủ và cập nhật tới nút D hay không. Nếu có nút I sẽ tạo ra một gói tin trả lời (route reply packet) và gửi ngược lại theo đường đi ngược vừa được tạo ra. Khi đó gói tin RREQ không cần phải tiếp tục lan truyền đi nữa. Trong trường hợp ngược lại, nếu không biết một đường đi tới D, nút I sẽ tiếp tục lan truyền gói tin RREQ. Trường TTL trong gói tin RREQ sẽ bị giảm đi một, và nếu nó bằng 0, việc lan truyền sẽ chấm dứt. Nếu vẫn tiếp tục được lan truyền, trường hop count trong gói tin sẽ được tăng lên 1.



Hình II-4.22 - Đường đi ngược được tạo ra khi RREQ lan truyền trong mạng

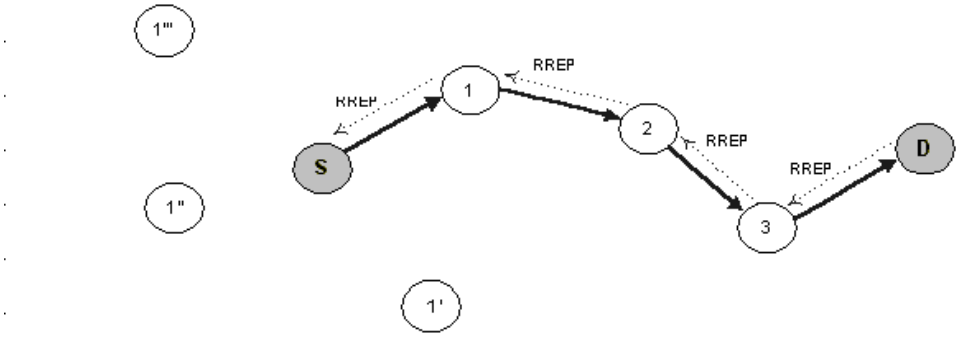
Tạo Route Reply

Khi một nút mạng biết một đường đi đầy đủ và cập nhật (hoặc nó chính là nút đích hoặc bảng định tuyến của nó có một thành phần tương ứng với nút đích), nó sẽ tạo một gói tin trả lời RREP gửi ngược lại tới nút nguồn.

Gói tin RREP sẽ chứa địa chỉ IP của cả nút nguồn và nút đích và số thứ tự của đường đi vừa tìm được. Gói tin này cũng có trường hop count và trường lifetime (thời gian sống) chứa giá trị tương ứng với thời gian có hiệu lực của đường đi (nếu sau khoảng thời gian đó đường đi không được sử dụng, nó sẽ bị xoá bỏ).

Chuyển tiếp Route Reply

Như trong hình 5, đường đi từ nút nguồn tới nút đích (forward path) được tạo ra khi gói tin trả lời được truyền theo đường đi ngược từ nút đích về nút nguồn. Mỗi nút nhận được gói tin RREP sẽ tạo ra trong bảng định tuyến của mình một thành phần tương ứng với nút đích D. Hai trường số thứ tự và khoảng cách trong thành phần mới tạo ra này sẽ nhận giá trị của số thứ tự đích và trường hop count trong gói tin RREP. Bước truyền tiếp theo là nút cuối cùng vừa chuyển tiếp gói tin RREP.



Hình II-4.23 - Đường đi từ nút nguồn đến nút đích được hình thành

Nếu gói tin RREP chưa tới được nút đích S, nó sẽ tiếp tục được chuyển tiếp dọc theo đường đi ngược đồng thời trường hop count trong gói tin cũng tự động tăng lên.

Khi gói tin RREP tới được nút nguồn, nút S sẽ tạo một thành phần trong bảng định tuyến tương ứng với nút đích D và tự động huỷ bỏ gói tin RREP. Tiến trình Discovery cũng kết thúc và một đường đi mới được thiết lập.

4.3.3.2 Quản lý kết nối cục bộ

Khi một đường đi tới nút đích được hình thành, mỗi nút mạng có thể sử dụng một số các kỹ thuật để giám sát trạng thái của đường đi đó. Nói một cách khác, mỗi nút nằm trên đường đi sẽ cố gắng đảm bảo bước truyền tiếp theo trên đường đi luôn trong trạng thái sẵn sàng. Nếu bước truyền tiếp theo ở trạng thái tích cực thì đường đi tương ứng vẫn có giá trị. Trong trường hợp ngược lại, nút hiện thời phải lập tức thông báo cho các nút nằm trước nó trên đường đi.

Quá trình giám sát có thể được thực hiện theo hai phương pháp khác nhau: proactive hoặc reactive. Đối với phương pháp proactive, nó sử dụng một số kiểu hoạt động đề phòng. Mỗi nút mạng sẽ liên tục giám sát trạng thái thực tế của các nút lân cận bằng cách cập nhật bản đồ kết nối cục bộ (local connectivity map). Bất cứ khi nào nhận được một gói tin thông báo (broadcast packet) gửi đến, nút hiện thời sẽ tiến hành cập nhật hoặc tạo mới một thành phần trong bảng định tuyến tương ứng với nút gửi thông báo. Thành phần này có thời gian sống - lifetime ngắn, tương ứng với khoảng thời gian lớn nhất mà một nút lân cận được cho phép giữ im lặng trước khi nút hiện thời cho rằng nút này đã mất giá trị. Do đó, khi một nút lân cận vẫn ở trạng thái tích cực, thành phần tương ứng với nó trong bảng định tuyến của nút hiện thời vẫn có giá trị. Trong trường hợp lifetime hết hạn mà không có thông báo mới, nút hiện thời sẽ cho rằng liên kết đã bị phá vỡ và chương trình thông báo lỗi liên kết (link failure notification procedure) lập tức được gọi. Tuy nhiên tình trạng "im lặng" của một nút có thể không phản ánh đúng trạng thái hiện tại của nó, do đó để đảm bảo tính chính xác, AODV còn sử dụng thêm một message "hello". Khi không nhận được các gói tin thông báo trạng thái của các liên kết, nút hiện thời sẽ gửi message "hello" đến các nút lân cận để kiểm tra tình trạng thật sự của các nút đó.

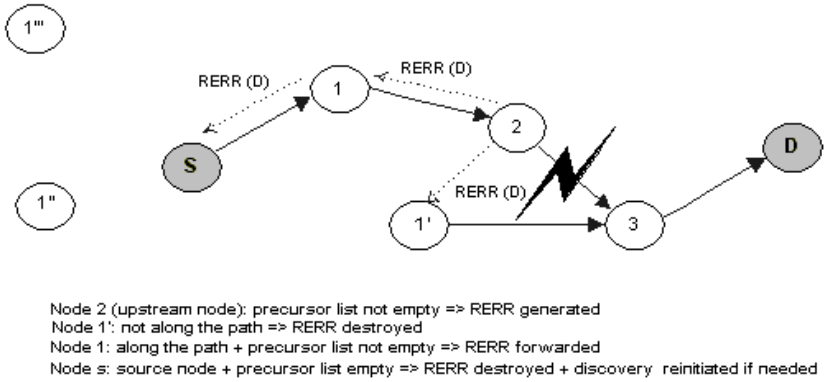
Phương pháp giám sát thứ hai được gọi là phương pháp proactive. Điều này có nghĩa là các liên kết bị phá vỡ chỉ bị phát hiện khi có sự cố khi truyền dữ liệu. Trong trường hợp này, lỗi kết nối được phát hiện muộn, điều này không phù hợp với mục đích của mạng MANET bởi nó rất dễ gây ra tình trạng quá tải cho mạng.

4.3.3.3 Duy trì đường đi

Khi một lỗi kết nối được phát hiện, các nút mạng sẽ lan truyền ngược gói tin báo lỗi route error (RERR). Gói tin RERR chứa danh sách các nút đích bị mất và số thứ tự tương ứng của chúng được tăng lên 1. Khi nhận được một gói tin RERR, các nút chịu ảnh hưởng sẽ cập nhật lại bảng định tuyến của nó.

Đối với mỗi nút đích trong gói tin RERR, nút hiện thời sẽ đặt lại giá trị cho trường khoảng cách thành vô cùng và cập nhật trường số thứ tự bằng cách sao chép số thứ tự của các nút tương ứng trong gói tin RERR. Ngoài ra, nếu trường precursor của nút hiện thời chưa rỗng, gói tin RERR sẽ tiếp tục được truyền ngược lại.

Chú ý rằng các nút sẽ chỉ cập nhật lại bảng định tuyến nếu gói tin RERR mà nó nhận được xuất phát từ bước truyền tiếp theo của nó trên đường đi tới nút đích. Ví dụ như trong hình 6, nút 1’ nhận được gói tin RERR thông báo đường đi tới nút D bị lỗi. Nhưng nút 1’ sẽ không cập nhật lại bảng định tuyến của nó (thực tế đường đi từ 1’ đến D không có sự cố) vì gói tin RERR đó được gửi từ nút 2, mà nút 2 không phải là bước truyền tiếp theo của nút 1’ trên đường đi tới nút D. Do đó ở nút 1’, gói tin RERR sẽ bị huỷ bỏ.



Hình II-4.24 - Lan truyền gói tin RERR

4.3.3.4 Thời gian hết hạn và việc huỷ bỏ một đường đi

Trong mô hình AODV nếu một đường đi không được sử dụng sau một thời gian nhất định nó sẽ bị huỷ bỏ. Do đó nếu một đường đi có trạng thái tích cực không được sử dụng trong khoảng thời gian cho phép - ACTIVE_ROUTE_TIMEOUT thì nút hiện tại sẽ huỷ bỏ nó bằng cách tăng số thứ tự của đường đi và thiết lập giá trị của trường hop count lên thành vô cùng. Tuy nhiên thành phần tương ứng với đường đi đó trong bảng định tuyến không bị xoá hẳn khỏi bảng, nó chỉ được đánh dấu hết hạn sử dụng. Trên thực tế, thành phần này vẫn tồn tại trong bảng thêm một khoảng thời gian DELETE_PERIOD trước khi bị xóa hoàn toàn.

Lý do đằng sau của việc có thêm một khoảng thời gian DELETE_PERIOD là để cho nút mạng có thể giữ được số thứ tự của đường đi đó ở mức lâu nhất có thể. Điều này giải thích cho việc tại sao khi một nút mạng gửi yêu cầu tìm đường, nó lại có thể xác định được số thứ tự đích lớn hơn số thứ tự của các đường đi đã tồn tại trước đó. Hơn nữa giả sử như một đường đi bị xoá ngay khi hết hạn thì giá trị của số thứ tự đích khi tạo yêu cầu tìm đường sẽ được đặt mặc định bằng 0, thực tế trường hợp này rất dễ gây ra hiện tượng vòng lặp.

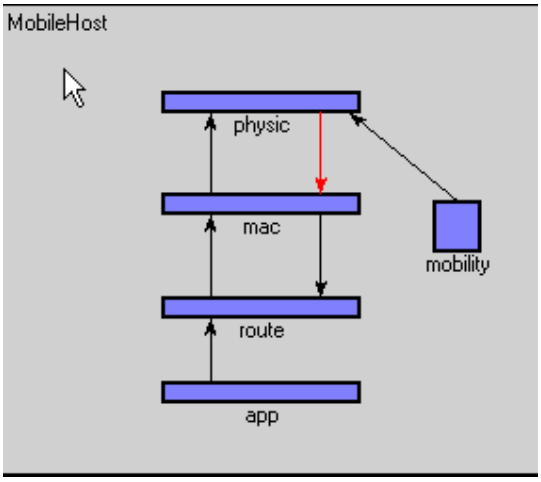
PHẦN III – PHÂN TÍCH THIẾT KẾ ỨNG DỤNG MÔ PHỎNG MẠNG ADHOC

1. MÔ HÌNH CHUNG

Hệ thống AdhocSim kèm theo báo cáo này bao gồm các tầng sau:

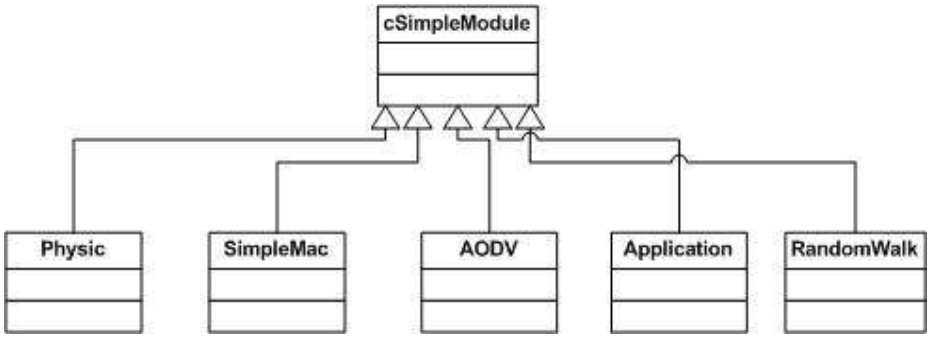
- Tầng vật lý (physical layer)
- Tầng điều khiển truy nhập (MAC layer)
- Tầng mạng (Route layer)
- Tầng ứng dụng (Application layer)
- Tầng di động (Mobility layer)

Mô hình các tầng



Hình III-1.1 - Mô hình mạng Adhoc

Mô hình các Class



2. CẤU TRÚC HỆ THỐNG

2.1 Tầng vật lý (Physical model)

Để thực hiện chức năng của tầng vật lý ở mỗi nút mạng, ta chủ yếu quan tâm đến việc tạo các cổng để cho phép trao đổi gói tin giữa các nút mạng.

Mỗi khi nút mạng di chuyển và tiến lại đủ gần (tùy thuộc vào năng lượng truyền dẫn – transmission power) tới một nút mạng lân cận, thì sẽ xảy ra các hoạt động sau:

1. Tạo một cổng mới cho các module kết hợp - compound module (bao gồm 2 nút mạng).

   Sử dụng hàm:

   int Physic::addNewGate(cModule *mod, char* gname, char type)

2. Tạo cổng trên mỗi physic module chứa trong mobile host module.
3. Tạo liên kết giữa cổng mới của module đơn giản (simple module) vừa được tạo ra và cổng mới của module kết hợp.

   void Physic::setUpConn(char kind,int& a,int& b)

4. Tạo liên kết giữa các module của 2 nút mạng. Đây là loại liên kết sử dụng kênh "etere". Kênh này được định nghĩa để mô phỏng môi trường truyền vô tuyến, với độ trễ, tỉ lệ lỗi ….

Chi tiết các tham số và cổng của một module kiểu Physic như sau:

```
simple Physic
    parameters:
        txPower : numeric,
        rxThreshold : numeric,
        channelDelay : numeric,
        channelDatarate : numeric,
        channelError : numeric;
    gates:
        in: fromMobility;
        in: fromMac;
        out: toMac;
endsimple
```

Mô phỏng hoạt động của PHY bằng các hàm sau:

- void Physic::initialize()
- void Physic::handleMessage(cMessage* msg)
- void Physic::detectNeighbours()
- void Physic::updateConnections()

2.2 Tầng điều khiển truy nhập (Mac Layer)

AdhocSim triển khai tầng MAC như sau:

```
simple Mac
    parameters:
        inBufferSize : numeric,
        promisqueMode : bool;
    gates:
        in: fromPh;
        in: fromRoute;
        out: toRoute;
        out: toPh;
endsimple
```

Giao thức chính được mô phỏng ở đây là CSMA/CA.

Tầng điều khiển truy nhập cho phép các message gửi đi nhưng đối với các message nhận vào nó sẽ kiểm tra trạng thái của các tầng ở trên. Nếu các tầng trên bận thì các message sẽ được đặt trong bộ đệm. Trong trường hợp bộ đệm đầy thì message đó sẽ bị hủy bỏ.

```
if(!buffer->empty())
                {

                        cMessage* m = buffer->pop();

                        send(m, "toRoute");

                        scheduleAt(elabTime() ,endService);
                }
                else
                {
                        d("no messages...waiting");
                        routerBusy = false;
                }
                else
                        {
                                if(buffer->canStore(msg->length()))
                                {
                                        d("host busy, will put the msg in the buffer!");
                                        buffer->insert(msg);
                                }
                                else
                                {
                                d("input buffer full!!! discarding pkts");
                                        bufferFullDiscard++;
                                        delete msg;
                                }
                        }
```

Tầng MAC sẽ kiểm tra địa chỉ MAC của các gói tin đến. Và chỉ cho qua các gói tin các địa chỉ MAC phù hợp hoặc các gói tin dạng broadcast.

```
if( ( (int)msg->par("mac") != parentModule()->id() ) &&
                  ( (int)msg->par("mac") != BROADCAST ) &&
                  ( ! promisqueMode) )
                {
                        d("message not for this node, discarding");
```

```
            delete msg;
        }
        else
        {
            //this node can handle the message
            d("got message from "<<msg->par("source"));
            //if the host was idle
            if( (!routerBusy) && (buffer->empty()) )
            {
                routerBusy = true;

                //this->takeOwnership(false)
                send(msg,"toRoute");
                //this->takeOwnership(true);
                scheduleAt( elabTime(), endService);
            }
```

2.3 Tầng mạng (Routing model)

Mô hình dẫn đường có thể coi là phần trọng tâm nhất trong mô hình Adhoc kèm theo báo cáo này. Hệ mô phỏng đi kèm báo cáo này đã thực hiện được các chức năng chính sau ở tầng mạng

- Trao đổi các gói tin HELLO giữa các nút mạng lân cận.
- Tìm kiếm theo vòng mở rộng (expanding ring search)
- Truyền tải các gói tin báo nhận (ACK message).
- Tạo một "black list" để tránh các nút mạng không đáng tin cậy. Tức là giả sử một nút X đưa nút lân cận Y trong black list, thì có nghĩa là X đã gửi gói tin RREQ mấy lần tới Y, nhưng X không nhận được bất kỳ tín hiệu ACK nào từ Y. Khi đó, X sẽ không tiếp nhận bất kỳ gói tin RREQ nào từ Y nữa. Tuy nhiên nó vẫn xử lý các gói tin Hello từ Y.

Tất cả các gói tin điều khiển (Control message) của AODV trong ứng dụng này đều có độ dài 512 byte. Trong khi đó các gói tin dữ liệu (Data message) có kích thước tùy theo các tham số thiết lập ở tầng ứng dụng.

Dưới đây là danh sách tên các loại message được sử dụng trong mô hình minh họa này.

HELLO	Hello message
RREQ	Route Request message (gói tin yêu cầu tìm đường)
RREP	Route Reply message (gói tin phản hồi tìm đường)
RERR	Route Error, chứa danh sách các nút đích không tới được.

DATA	Data message
RREP ACK	RREP acknowledgment message (gói tin báo nhận RREP)
DELETE	Gói tin của bản thân nút mạng (self message) để kích hoạt trigger sự kiện một đường đi đã hết hạn sử dụng.
FLUSH	Gói tin thực hiện RREP time out
SEND HELLO	Gói tin của bản thân nút mạng (self message) để kích hoạt nút mạng đó gửi gói tin Hello.
BLK LIST	Gói tin của bản thân nút mạng (self message) để kích hoạt sự kiện rút một nút mạng lân cận khỏi "black list".

Mỗi loại message đều có các tham số để trao đổi thông tin điều khiển giữa các nút mạng. Chương trình mô phỏng AdhocSim cũng chứa hầu hết các tham số trong các gói tin của một mạng Adhoc chuẩn. Như là: dest, seqNumS, seqNumD, Source, mac, ttl, hopNum …..

Mô tả topo tầng mạng

```
simple Routing
    gates:
        in: fromMac;
        in: fromApp;
        out: toMac;
endsimple

//IEEE-802.11a channel
//channel etere
//      delay 0.0001;
//      datarate 11.04858e+6 ; // 11Mbit/sec
//      error 0.000001;
//endchannel
```

Quá trình xử lý các loại gói tin trong bảng trên

```
switch(msg->kind())
        {
        case SEND_HELLO:
                d("sendHello");
                reply = generateHELLOmsg();
                broadcast(reply);
                break;

        case DELETE:
                d("delete");
                reply = handleDelete(msg);
                broadcast(reply);
```

```
            break;

        case HELLO:
            d("hello");
            handleHELLO(msg);
            delete msg;
            break;

        case FLUSH:
            d("flush");
            reply = handleFlush(msg);
            broadcast(reply);
            break;

        case RREQ:
            d("rreq "<<msg->name());
            reply = handleRREQ(msg);
            //if the message received need a reply
            //then send it to the mac module that will
            //care about sending it around
            broadcast(reply);
            delete msg;
            break;

        case RREP:
            d("rrep");
            reply = handleRREP(msg);
            broadcast(reply);
            delete msg;
            break;

        case RERR:
            d("rerr");
            reply = handleRERR(msg);
            broadcast(reply);
            delete msg;
            break;

        case DATA:
            d("data");
            reply = handleData(msg);
            broadcast(reply);
            delete msg;
            break;

        case RREP_ACK:
            d("ack");
            handleACK(msg);
            delete msg;
            break;
```

```
        case ESP_ACK:
                d("esp_ack");
                reply = handleESP_ACK(msg);
                broadcast(reply);
                break;

        case BLK_LIST:
                d("black list");
                handleBLK_LIST(msg);
                //delete msg;
                break;
        }
}
```

Các hàm tương ứng được gọi:

cMessage* AODV::generateHELLOmsg()

void AODV::handleHELLO(cMessage* msg)

cMessage* AODV::checkRouteTable(RouteTableElement*b,cMessage* reply,int& errors)

cMessage* AODV::handleFlush(cMessage* msg)

cMessage* AODV::handleDelete(cMessage* msg)

cMessage* AODV::handleData(cMessage* msg)

RouteTableElement* AODV::findNode(int n)

cMessage* AODV::handleRERR(cMessage *msg)

cMessage* AODV::handleRREP(cMessage *msg)

void AODV::handleACK(cMessage* msg)

cMessage* AODV::handleRREQ(cMessage *msg)

2.4 Mobility models

```
simple Mobility
    parameters:
        minSpeed : numeric, // vel in m/s
        maxSpeed : numeric,
        movKind : numeric, //0 torus, 1 rebound
        XRange : numeric,
        YRange : numeric,
        pauseTime : numeric, //pause waited after reacing of
        moveInterval : numeric, //time to elaps between two
        distance : numeric, //used in random waypoint
        target : numeric, //used in pursuit model
        speedVariator : numeric;
    gates:
        out: out;
endsimple
```

Trong ứng dụng AdhocSim này, ta sử dụng Random Walk mobility model. Kiểu mô hình được chỉ ra trong file omnetpp.ini.

#mobility model
;include Ini/avrSpeed.ini
;world.mobileHost[*].mobilityModel = "RandomWalk"
;include Ini/randWalk.ini
;world.mobileHost[*].mobilityModel = "RestrictedRandWalk"
;include Ini/resRWalk.ini
world.mobileHost[*].mobilityModel = "RandomWP"
include Ini/randWP.ini
;world.mobileHost[*].mobilityModel = "RandomDirection"
;include Ini/rDir.ini
;world.mobileHost[*].mobilityModel = "Pursuit"
;include Ini/pursuit.ini
;world.mobileHost[*].mobilityModel = "Normal"
;include Ini/normal.ini

Hoạt động của RandomWalk được triển khai trong hàm

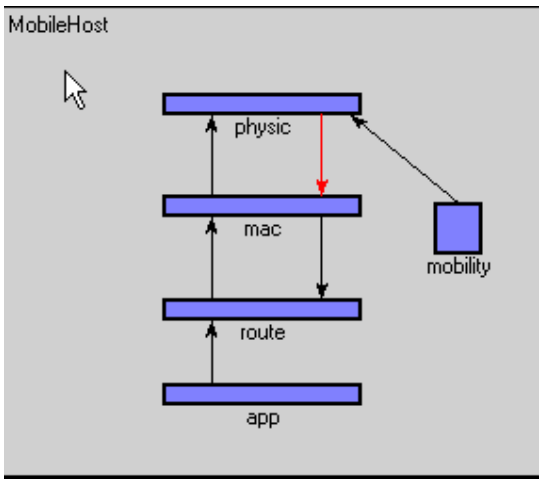
double RandomWalk::randomWalk(int& x, int& y)

2.5 Tầng ứng dụng

```
simple Application
    parameters:
        rate : numeric, //paket per secod
        pktSize : numeric,
        hostNum : numeric,
        active : numeric,
        burstInterval : numeric; // time(s) between two data
    gates:
        out: out;
endsimple
```

2.6 Liên kết giữa các tầng

File “mobilehost.ned” chứa khai báo liên kết giữa các tầng của mô hình AdhocSim.



Liên kết giữa tầng vật lý và tầng MAC

```
physic.toMac <--> mac.fromPh

mac.toPh --> physic.fromMac
```

Liên kết giữa tầng MAC và tầng dẫn đường

```
mac.toRoute --> route.fromMac

route.toMac --> mac.fromRoute
```

Liên kết giữa tầng dẫn đường và tầng ứng dụng

```
app.out --> route.fromApp
```

Liên kết giữa tầng mobility và tầng PHY

```
mobility.out --> physic.fromMobility
```

2.7 Thiết lập các thông số cho hệ mô phỏng

Các thông số được thiết lập trong file “omnetpp.ini”

2.7.1 Thông số của Map và Hosts

Vùng phủ sóng	700m x 700m
Số lượng nút mạng	25
Host enabled to transmit	5

Đoạn code tương ứng trong file omnetpp.ini

world.height = 700

world.width = 700

world.dim = 25

2.7.2 Physical Layer

Transmission power	25000ρWatt
Receive Threshold	1ρWatt
Channel Bandwidth	11Mb/s (IEEE 802.11a)
Channel Delay	10 μ sec
Channel Error probability	1 bit on 106

Đoạn code tương ứng trong file omnetpp.ini

#physic module

world.mobileHost[*].physic.txPower = uniform(9000,9900)

world.mobileHost[*].physic.rxThreshold = 1

world.mobileHost[*].physic.channelDelay = 0.0001

world.mobileHost[*].physic.channelDatarate = 11.04858e+6

world.mobileHost[*].physic.channelError = 0.000001

2.7.3 Mac Layer

Busy time	các tham số λ và γ
λ	15 μ sec
γ	7.5 μ sec
Input Buffer size	1MB

Đoạn code tương ứng trong file omnetpp.ini:

world.mobileHost[*].mac.promisqueMode = true;

world.mobileHost[*].mac.inBufferSize = 8.38864e6

2.7.4 Routing

Control Message Size	64 byte
HELLO interval	1sec
Allowed HELLO loss	2
Delete pertiod	4sec
RREQ max trials	3

2.7.5 Application

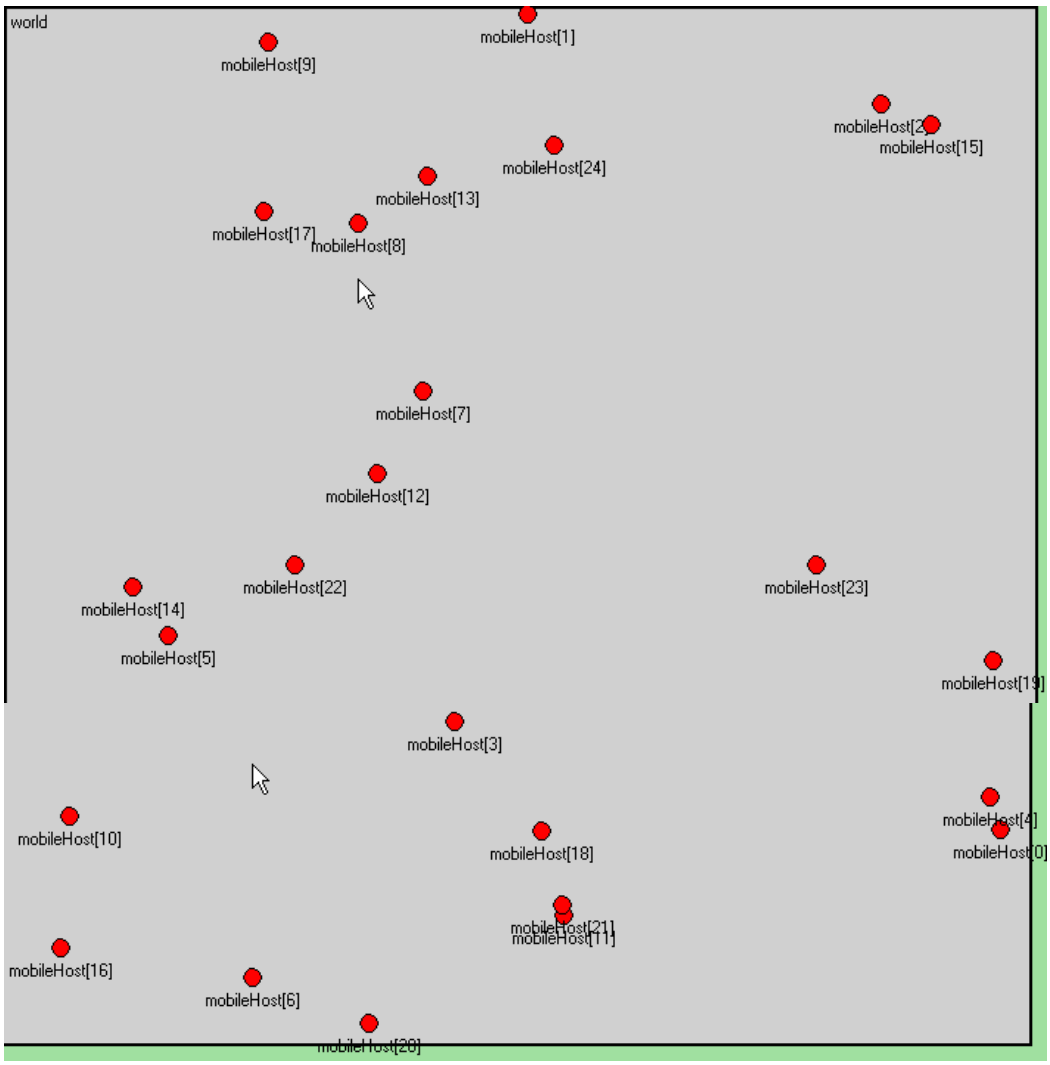
Enabled Node	5 (nút số 1,3,4,7,10)
Message packet size	512byte (4096 bits)
Burst length	64 packets
Send Packet Rate	3/sec
Burst Interval	phân phối đều trong đoạn [0.1,3]sec

Đoạn code tương ứng trong file omnetpp.ini:

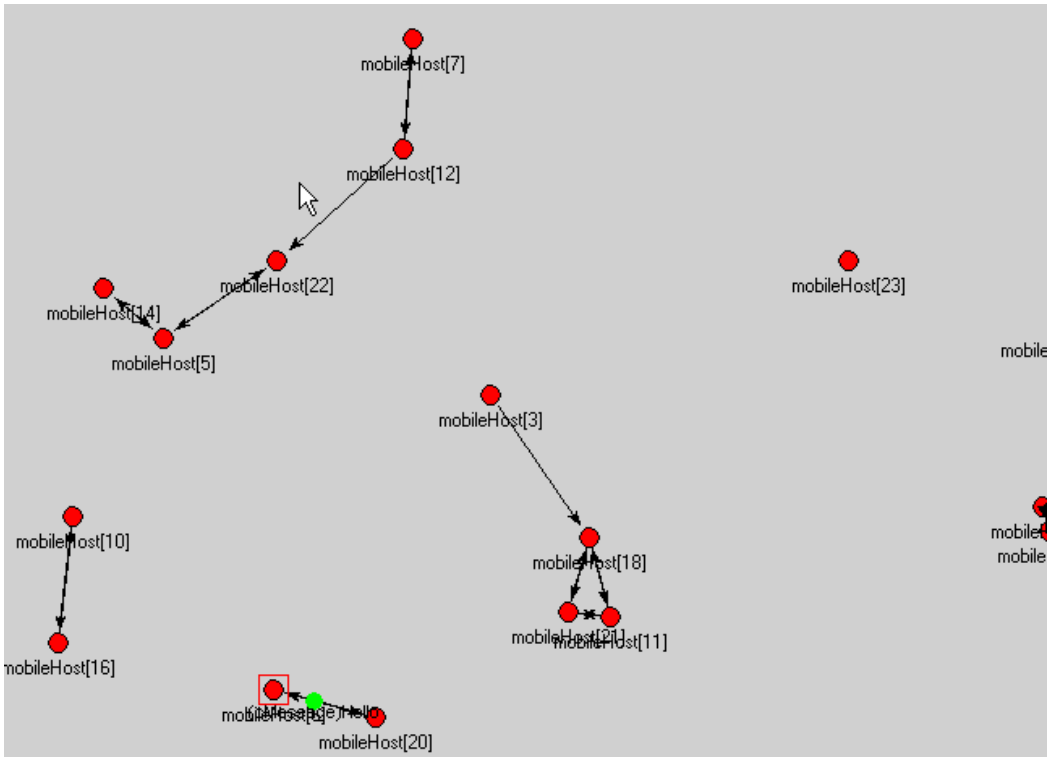
```
;pakets per second
world.mobileHost[*].app.rate = 3
;pakets of 512 byte = 4096 bit
world.mobileHost[*].app.pktSize = 4096
;time elapsed  between two data burst
world.mobileHost[*].app.burstInterval = truncnormal(2,1.0)
;Enabled Node = 5
world.mobileHost[1].app.active = 1
world.mobileHost[3].app.active = 1
world.mobileHost[7].app.active = 1
world.mobileHost[10].app.active = 1
world.mobileHost[4].app.active = 1
```

3. KẾT QUẢ THỰC HIỆN

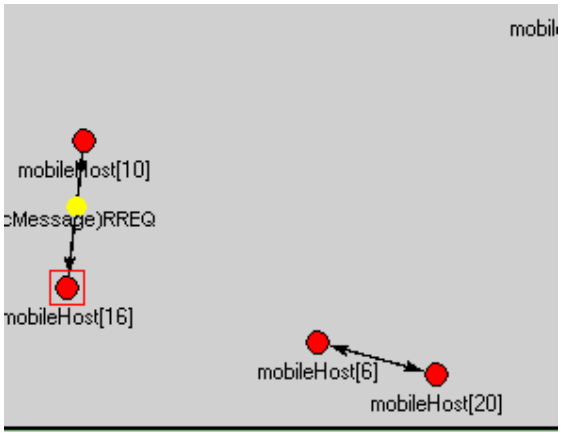
3.1 Topo



3.2 Gửi các gói tin Hello



3.3 Gửi gói tin RREQ



Phần IV - PHỤ LỤC

1. SO SÁNH OMNET++ VÀ NS-2

	OMNet++	NS-2
Tính linh hoạt	OMNeT++ là một công cụ mô phỏng rất linh hoạt, có thể mô phỏng bất kỳ một hệ thống nào hoạt động dựa trên nguyên tắc trao đổi các gói tin giữa các thành phần tích cực. Ví dụ như các kiểu mạng hàng đợi, hệ thống vi xử lý, kiến trúc phần cứng...	ns-2 là một công cụ được thiết kế dành riêng để mô phỏng mạng (TCP/IP). Rất khó có thể sử dụng ns-2 cho những hệ thống khác (những hệ thống không sử dụng nguyên tắc mạng chuyển mạch gói).
Mô hình lập trình	OMNeT++ sử dụng ngôn ngữ lập trình hướng đối tượng C++. OMNeT++ đồng thời hỗ trợ cả hai kiểu, tạo file bằng cách sử dụng các trình soạn thảo thông thường hoặc sử dụng chương trình GNED có giao diện đồ hoạ trực quan có sẵn.	Cũng sử dụng ngôn ngữ C++
Quản lý mô hình	Phần nhân mô phỏng của OMNeT++ là một lớp thư viện. Các mô hình được tạo ra hoàn toàn độc lập với phần nhân mô phỏng. Người lập trình có thể viết các thành phần dựa vào các lớp thư viện khác, sau đó biên dịch và liên kết nó với thư viện mô phỏng của OMNeT++. Điều này có nghĩa là phần code của OMNeT++ không cần phải thay đổi lại khi thiết kế một mô hình (các lớp trong OMNeT++ có khả năng sử dụng lại cao).	Tính sử dụng lại trong ns-2 không cao, để thêm vào một lớp mới, bạn cần phải download toàn bộ mã nguồn về, sửa đổi lại cho phù hợp với mô hình của mình, copy thêm các file của bạn vào, gán thêm các giá trị khác...
Hỗ trợ mô hình quan hệ kế thừa	Trong OMNeT++ các module có quan hệ kế thừa. Điều này không những tạo sự dễ dàng cho người lập trình khi mô phỏng những hệ thống phức tạp mà nó cũng làm tăng khả năng sử dụng	Trong ns-2, các mô hình có quan hệ ngang bằng. Việc tạo một mạng con từ một mạng lớn hay thực hiện một giao thức phức tạp từ những đơn vị độc lập nhỏ là không thể thực hiện được.

	lại cho các lớp của OMNeT++.	
Tài liệu trợ giúp	Những tài liệu trợ giúp của OMNeT++ được cập nhật thường xuyên và rất đầy đủ. Đặc biệt có phần Tutorial và API rất hữu ích đối với người sử dụng.	Tài liệu trợ giúp của ns-2 không có tính liền mạch (không có phần tutorial). Không có sự phân biệt rõ ràng giữa một mô hình với các thư viện mô phỏng trong ns-2.
Mô phỏng cácc mạng lớn	OMNeT++ có thể mô phỏng các mạng có cấu trúc cực lớn. Giới hạn của nó phụ thuộc vào kích thước bộ nhớ ảo của máy sử dụng.	ns-2 làm việc không tốt trong trường hợp mô phỏng các mạng có cấu trúc lớn.
Thiết kế thử nghiệm	Tham số của các thiết kế thử nghiệm được khai báo trong file cấu hình omnetpp.ini và nó có thể được thay đổi dễ dàng phục vụ cho mục đích thử nghiệm.	Trong ns-2, các tham số được gắn trực tiếp trong Tcl script do đó khó có thể thay đổi chúng. Điều này không có lợi cho những mô hình được thiết kế với mục đích thử nghiệm.

2.TÀI LIỆU THAM KHẢO

- http://omnetpp.org
- Y. Ahmet S¸ekercioglu Simulation of IPv6 Networks with OMNeT++
- Jeroen Idserda TCP/IP modelling in OMNeT++
- Simulating Wireless Sensor Networks with OMNeT++ C. Mallanda, A. Suri, V. Kunchakarra, S.S. Iyengar*, R. Kannan* and A. Durresi Sensor Network Research Group, Department of Computer Science, Louisiana State University, Baton Rouge, LA.