Bài giảng
LẬP TRÌNH MẠNG

Người soạn: Lương Ánh Hoàng

Bộ môn Kỹ thuật máy tính

Viện Công nghệ thông tin và Truyền thông, ĐHBK Hà nội

Hà nội, 8/2010

Mục lục bài giảng

Chương 1. Giới thiệu các mô hình lập trình mạng

Bài giảng số 1

- **Thời lượng: 3 tiết.**
- **Tóm tắt nội dung :**
  - **Định nghĩa lập trình mạng.**
  - **Ứng dụng của lập trình mạng.**
  - **Các ngôn ngữ lập trình.**
  - **Các thư viện và môi trường hỗ trợ lập trình mạng**
  - **Giao thức IP.**

1.1 Tổng quan về lập trình mạng

1.1.1 Khái niệm

Lập trình mạng là các kỹ thuật lập trình nhằm xây dựng những ứng dụng, phần mềm khai thác hiệu quả tài nguyên mạng máy tính.
Mạng máy tính đang ngày một phát triển, ứng dụng của mạng đem lại là không thể phủ nhận. Giáo trình này sẽ đề cập đến một vài phương pháp xây dựng các ứng dụng tận dụng được hạ tầng mạng sẵn có.

1.1.2 Ngôn ngữ lập trình

Hầu hết các ngôn ngữ lập trình đều có thể sử dụng để lập trình mạng, tuy nhiên việc lập trình mạng còn phụ thuộc vào các thư viện và môi trường lập trình có hỗ trợ hay không. Có thể liệt kê các ngôn ngữ lập trình có thể sử dụng để lập trình mạng như sau:

- C/C++: Ngôn ngữ lập trình rất mạnh và phổ biến, dùng để viết mọi loại ứng dụng trong đó có ứng dụng mạng.
- Java: Ngôn ngữ lập trình khá thông dụng và hỗ trợ trên nhiều môi trường, trong đó có thể viết ứng dụng chạy trên điện thoại di động.
- C#: Ngôn ngữ lập trình cũng rất mạnh và dễ sử dụng, chỉ hỗ trợ trên họ hệ điều hành Windows của Microsoft.
- Python, Perl, Php…: Các ngôn ngữ thông dịch, sử dụng để viết nhanh các tiện ích nhỏ một cách nhanh chóng, trong đó có thể sử dụng để viết ứng dụng mạng.

Học phần này sẽ trình bày phương pháp lập trình mạng dựa trên hai ngôn ngữ: C/C++ và C#.

1.1.3 Thư viện hỗ trợ

Việc lập trình mạng phụ thuộc rất nhiều vào các thư viện hỗ trợ đến từ hệ thống. Tùy thuộc vào nền tảng phát triển ứng dụng mà có thể sử dụng các thư viện khác nhau. Có thể liệt kê một vài thư viện hỗ trợ lập trình mạng như sau:

- Winsock: Thư viện liên kết động của Microsoft, được phân phối cùng hệ điều hành Windows. Winsock cung cấp khá nhiều API để phát triển ứng dụng mạng. Winsock có thể sử dụng cùng bất kỳ ngôn ngữ lập trình nào, nhưng bộ đôi C/C++ và Winsock đem lại hiệu năng cao nhất, nhưng tương đối khó sử dụng.
- Thư viện System.Net trong .NET framework: Thư viện cung cấp rất nhiều API dễ sử dụng để xây dựng ứng dụng mạng. Để sử dụng thư viện này, người ta thường dùng C#. Việc phát triển ứng dụng mạng nhờ thư viện này khá dễ dàng.
- Thư viện MFC Socket: Thư viện đi cùng bộ phát triển Visual Studio C++. Đây là thư viện cũng khá dễ sử dụng.
- Các thư viện trong Java Runtime, PHP,….

Giáo trình này sẽ trình bày cách sử dụng ba thư viện Winsock, System.Net và MFC Socket.

1.2 Giao thức Internet

Giao thức Internet (IP – Internet Protocol) là giao thức mạng thông dụng nhất trên thế giới. Thành công của Internet phần lớn là nhờ vào IPv4. IP được cài đặt rộng rãi trên hầu hết các hệ điều hành, trong các mạng nội bộ, các mạng diện rộng và Internet. Sự bùng nổ về số lượng máy tính cá nhân dẫn đến IPv4 càng trở nên hạn chế, đó là tiền đề cho việc phát triển giao thức mạng mới: IPv6. Chúng ta sẽ nhắc lại kiến thức cơ bản về Internet trong chương 2.

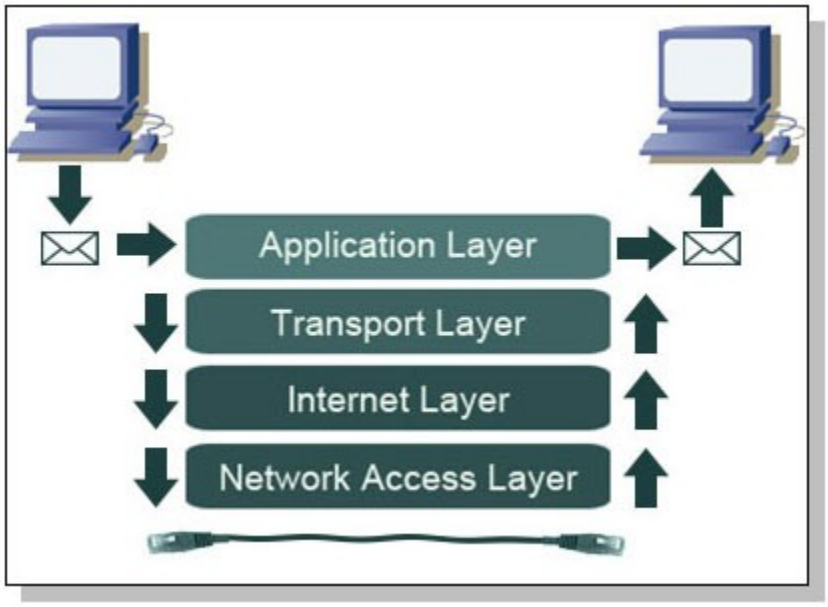
Chương 2. Bộ giao thức Internet (TCP/IP)

Bài giảng số 2

- **Thời lượng: 3 tiết.**
- **Tóm tắt nội dung :**
  - **Nhắc lại về bộ giao thức TCP/IP.**
  - **Giao thức IP.**
  - **Địa chỉ IPv6.**
  - **Giao thức TCP và UDP.**
  - **Dịch vụ DNS.**

2.1 Giới thiệu

Bộ giao thức TCP/IP, ngắn gọn là TCP/IP (tiếng Anh: Internet protocol suite hoặc IP suite hoặc TCP/IP protocol suite - bộ giao thức liên mạng), là một bộ các giao thức truyền thông cài đặt tầng giao thức mà Internet và hầu hết các mạng máy tính thương mại đang chạy trên đó. Bộ giao thức này được đặt tên theo hai giao thức chính của nó là TCP (Giao thức Điều khiển Giao vận) và IP (Giao thức Liên mạng). Chúng cũng là hai giao thức đầu tiên được định nghĩa.



Hình 1 . Các tầng giao thức TCP/IP

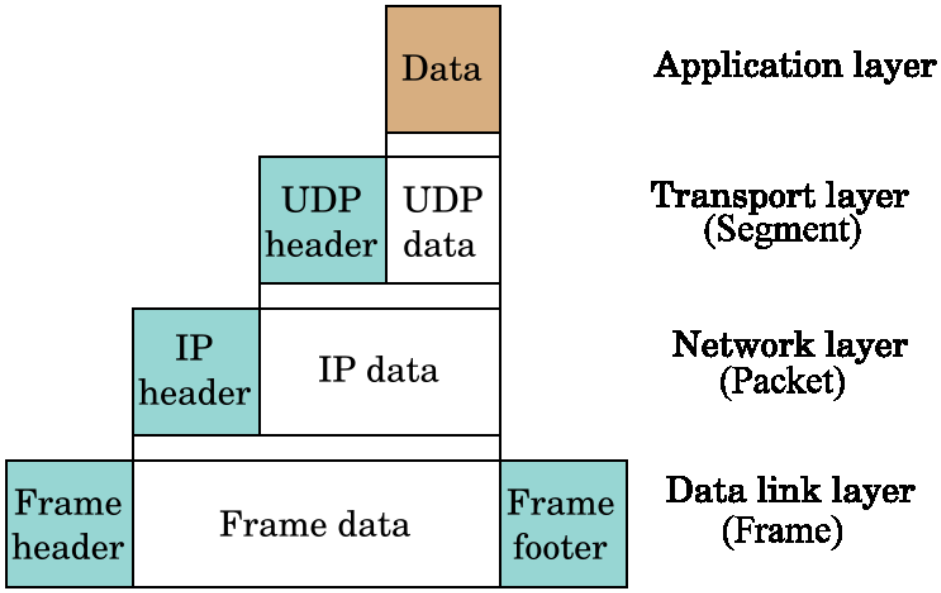
TCP/IP là một thể hiện thực tế của mô hình OSI. Mô hình OSI nguyên bản chia thành 7 tầng giao thức, tuy nhiên TCP/IP hiện thực và đơn giản hóa đi chỉ còn 4 tầng.

- Tầng ứng dụng (Application Layer): Bao gồm các giao thức đóng gói dữ liệu từ ứng dụng, người dùng rồi truyền xuống tầng thấp hơn. Các giao

thức có thể kể đến ở tầng này là: HTTP, FTP, Telnet, DNS, SSH, SMTP, POP3,…

- Tầng giao vận (Transport Layer): Nhận dữ liệu từ ứng dụng tầng trên và thông qua tầng dưới, truyền dữ liệu tới ứng dụng ở máy tính đích. Tầng này cung cấp dịch vụ truyền dữ liệu giữa ứng dụng - ứng dụng. Các giao thức ở tầng này: TCP, UDP, ICMP.
- Tầng liên mạng (Network Layer): Định tuyến và truyền gói tin liên mạng. Tầng này cung cấp dịch vụ truyền dữ liệu là các gói tin giữa các nút mạng trong cùng một mạng hoặc liên mạng. Các giao thức ở tầng này: IPv4, IPv6,…
- Tầng liên kết (Network Acess Layer, Datalink Layer): Truyền dữ liệu giữa các nút mạng trên cùng một nhánh mạng. Tầng này làm việc trực tiếp với thiết bị chịu trách nhiệm chuyển đổi các bit sang một dạng tín hiệu vật lý khác (ánh sáng, điện, điện từ…)

Dữ liệu của người dùng sẽ lần lượt đi qua các tầng trong mô hình, ở mỗi tầng, dữ liệu sẽ được thêm phần header để điều khiển và chuyển xuống tầng thấp hơp. Bên nhận sẽ lần lượt bóc tách các header từ tầng thấp và chuyển lên tầng cao, cho đến người dùng. Hình dưới đây minh họa quá trình đóng gói dữ liệu của một ứng dụng sử dụng giao thức UDP.



Hình 2: Đóng gói dữ liệu UDP

2.2 IPv4

Giao thức Internet phiên bản 4 (IPv4) là phiên bản thứ tư trong quá trình phát triển của các giao thức Internet (IP). Đây là phiên bản đầu tiên của IP được sử dụng rộng rãi. IPv4 cùng với IPv6 (giao thức Internet phiên bản 6) là nòng cốt của giao tiếp internet. Hiện tại, IPv4 vẫn là giao thức được triển khai rộng rãi

nhất trong bộ giao thức của lớp internet. Các giao thức IP chạy ở tầng liên mạng.
Giao thức này được công bố bởi IETF trong phiên bản RFC 791 (tháng 9 năm 1981), thay thế cho phiên bản RFC 760 (công bố vào tháng giêng năm 1980). Giao thức này cũng được chuẩn hóa bởi bộ quốc phòng Mỹ trong phiên bản MIL-STD-1777.
IPv4 là giao thức hướng dữ liệu, được sử dụng cho hệ thống chuyển mạch gói (tương tự như chuẩn mạng Ethernet). Đây là giao thức truyền dữ liêu hoạt động dựa trên nguyên tắc tốt nhất có thể, trong đó, nó không quan tâm đến thứ tự truyền gói tin cũng như không đảm bảo gói tin sẽ đến đích hay việc gây ra tình trạng lặp gói tin ở đích đến. Việc xử lý vấn đề này dành cho tầng trên của bộ giao thức TCP/IP. Tuy nhiên, IPv4 có cơ chế đảm bảo tính toàn vẹn dữ liệu thông qua sử dụng trường checksum.

2.2.1 Địa chỉ IPv4

IPv4 sử dụng 32 bits để đánh địa chỉ, theo đó, số địa chỉ tối đa có thể sử dụng là 4,294,967,296 (232). Tuy nhiên, do một số được sử dụng cho các mục đích khác như: cấp cho mạng cá nhân (xấp xỉ 18 triệu địa chỉ), hoặc sử dụng làm địa chỉ quảng bá (xấp xỉ 16 triệu), nên số lượng địa chỉ thực tế có thể sử dụng cho mạng Internet công cộng bị giảm xuống. Với sự phát triển không ngừng của mạng Internet, nguy cơ thiếu hụt địa chỉ đã được dự báo, tuy nhiên, nhờ công nghệ NAT (Network Address Translation - Chuyển dịch địa chỉ mạng) tạo nên hai vùng mạng riêng biệt: Mạng riêng và Mạng công cộng, địa chỉ mạng sử dụng ở mạng riêng có thể dùng lại ở mạng công công mà không hề bị xung đột, qua đó trì hoãn được vấn đề thiếu hụt địa chỉ.

Địa chỉ IPv4 được chia làm 4 nhóm, mỗi nhóm 8 bit (octet) và được biểu diễn dưới dạng thập phân hoặc thập lục phân . Thí dụ:

Dạng biểu diễn	Giá trị
Nhị phân	11000000.10101000.00000000.00000001
Thập phân	192.168.0.1
Thập lục phân	0xC0A80001

2.2.2 Các lớp địa chỉ

Dải địa chỉ IPv4 được chia thành các lớp địa chỉ con. Có 5 lớp lớn A,B,C,D,E.

Lớp	MSB	Địa chỉ đầu	Địa chỉ cuối

A	0xxx	0.0.0.0	127.255.255.255
B	10xx	128.0.0.0	191.255.255.255
C	110x	192.0.0.0	223.255.255.255
D	1110	224.0.0.0	239.255.255.255
E	1111	240.0.0.0	255.255.255.255

Địa chỉ lớp A,B,C được sử dụng để trao đổi thông tin thông thường, địa chỉ lớp D sử dụng trong Multicast và lớp E chưa được sử dụng và để dành riêng sau này.

2.2.3 Mặt nạ mạng

Địa chỉ IPv4 được chia làm hai phần: phần mạng và phần host. Các bộ định tuyến sẽ sử dụng phần mạng để chuyển tiếp gói tin tới mạng đích. Thí dụ

Network	**Host**
192.168.0.	1
11000000.10101000.00000000.	00000001

Bảng :Phần mạng và phần host

Mặt nạ mạng được sử dụng để đánh dấu phần mạng và phần host. Có ba cách biểu diễn mặt nạ mạng:

- Biểu diễn dưới dạng **/n**, trong đó n là số bit dành cho phần mạng

  Thí dụ: 192.168.0.1/24.
- Biểu diễn dưới dạng nhị phân: dùng 32 bit để đánh dấu, trong đó các bit dành cho phần mạng là 1, các bit dành cho phần host là 0.
  Thí dụ: 11111111.11111111.11111111.00000000
  hay 255.255.255.0
- Biểu diễn dưới dạng Hexa: dùng số hexa để biểu diễn, tương tự như dạng nhị phân. Cách này ít được sử dụng.
  Thí dụ: 0xFFFFFF00

Với mỗi mạng có n bit dành cho phần mạng, thì sẽ có 32-n bit dành cho phần host. Phân phối địa chỉ trong mạng đó như sau

- 01 địa chỉ mạng (các bit phần host bằng 0).
- 01 địa chỉ quảng bá (các bit phần host bằng 1).
- 2^n – 2 địa chỉ còn lại có thể gán cho các máy trạm.

Thí dụ với địa chỉ 192.168.0.1/24:

- Địa chỉ mạng: 192.168.0.0
- Địa chỉ quảng bá: 192.168.0.255

- Địa chỉ host: 192.168.0.1 – 192.168.0.254

2.2.4 Các dải địa chỉ đặc biệt

Các dải địa chỉ đặc biệt, không được sử dụng trên Internet

Địa chỉ	**Diễn giải**
10.0.0.0/8	Mạng riêng
127.0.0.0/8	Địa chỉ loopback
172.16.0.0/12	Mạng riêng
192.168.0.0/16	Mạng riêng
224.0.0.0/4	Multicast
240.0.0.0/4	Dự trữ

Trong khoảng 4 tỉ địa chỉ có thể sử dụng của IPv4, người ta dành riêng ra ba dải địa chỉ để sử dụng trong các mạng nội bộ, các địa chỉ nội bộ sẽ chỉ sử dụng để trao đổi thông tin nội bộ trong mạng, và không có ý nghĩa trên Internet. Để kết nối mạng nội bộ với Internet, người ta dùng thiết bị gọi là NAT, NAT sẽ chuyển đổi địa chỉ nội bộ sang một địa chỉ toàn cục đại diện cho cả mạng, đi ra ngoài Internet.

Tên	Dải địa chỉ	Số lượng địa chỉ	Mô tả mạng	Viết gọn
Khối 24-bit	10.0.0.0–10.255.255.255	16,777,216	Một dải trọn vẹn thuộc lớp A	10.0.0.0/8
Khối 20-bit	172.16.0.0–172.31.255.255	1,048,576	Tổ hợp từ mạng lớp B	172.16.0.0/12
Khối 16-bit	192.168.0.0–192.168.255.255	65,536	Tổ hợp từ mạng lớp C	192.168.0.0/16

2.3 IPv6

Giao thức IPv6 là phiên bản tiếp theo của IP, được thiết kế dựa trên thành công IPv4, phiên bản vẫn còn được sử dụng rộng rãi hiện nay. IPv6 là giao thức trong tầng liên mạng trong mạng chuyển mạch gói TCP/IP. Động lực chính để thúc đẩy sự ra đời của IPv6 là do cạn kiệt tài nguyên địa chỉ IPv4. IPv6 được giới thiệu năm 1998 bởi IETF (Internet Engineering Task Force). Giáo trình này sẽ chỉ đề cập đến IPv4.

2.4 TCP

Transmission Control Protocol – TCP là một giao thức lõi chạy ở tầng giao vận, cung cấp các dịch vụ truyền dữ liệu theo dòng, tin cậy và được sử dụng bởi hầu hết các ứng dụng hiện nay. TCP chạy bên trên IP và chạy bên dưới ứng dụng. Việc lập trình mạng sẽ chủ yếu sử dụng giao thức này để truyền dữ liệu.

Giao thức IP cung cấp cơ chế truyền dữ liệu là các gói tin giữa các máy với nhau, nhưng không có sự đảm bảo về trật tự, mất mát thông tin. Trái lại, TCP cung cấp dịch vụ truyền dữ liệu chính xác, theo dòng, và đúng trật tự giữa các ứng dụng trên các máy khác nhau. Ngoài ra TCP còn kiểm soát tốc độ truyền, chống nghẽn mạng…

TCP thực hiện chia dữ liệu từ tầng ứng dụng thành các đoạn, mỗi đoạn kích thước thường không vượt quá kích thước của gói tin IP. TCP thêm các thông tin điều khiển vào phần đầu đoạn và chuyển xuống tầng dưới để gửi đi. Dữ liệu của các ứng dụng trên cùng một máy tính được phân biệt thông qua trường Port (16 bit) trong header của TCP. Nếu nột ứng dụng muốn nhận thông tin từ mạng, nó sẽ đăng ký một cổng với hệ điều hành, và TCP sẽ chuyển dữ liệu tới ứng dụng đó.

TCP Header																															
Bit offset	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
0	Source port																Destination port															
32	Sequence number																															
64	Acknowledgment number																															
96	Data offset			Reserved				CWR	ECE	URG	ACK	PSH	RST	SYN	FIN		Window Size															
128	Checksum																Urgent pointer															
160 …	Options (if Data Offset > 5) …																															

Hình : TCP header

Có rất nhiều dịch vụ ở tầng ứng dụng sử dụng TCP, thí dụ dịch vụ web chạy ở cổng 80, FTP ở cổng 21, SMTP cổng 25, POP3 cổng 110, IMAP cổng 143…

2.5 UDP

UDP – User Datagram Protocol, cũng là một giao thức lõi trong bộ TCP/IP. UDP cung cấp cơ chế truyền dữ liệu giữa các ứng dụng trên các máy khác nhau. UDP thực hiện chia nhỏ dòng dữ liệu ở tầng ứng dụng thành các đơn vị gọi là datagram và chuyển xuống tầng mạng. Tuy nhiên giống với IP, UDP không đảm

bảo thứ tự của các datagram, cũng như cơ chế phát hiện sự mất mát lỗi hoặc trùng datagram. Dù vậy UDP hoạt động tương đối nhanh và hiệu quả với những thông điệp ngắn và yêu cầu khắt khe về mặt thời gian.

UDP thích hợp với những ứng dụng cần tính thời gian thực cao, có thể sai sót như thoại, video…

UDP cũng sử dụng một số 16 bit trong header gọi là cổng để phân biệt giữa các ứng dụng. Cấu trúc UDP header đơn giản hơn TCP nhiều.

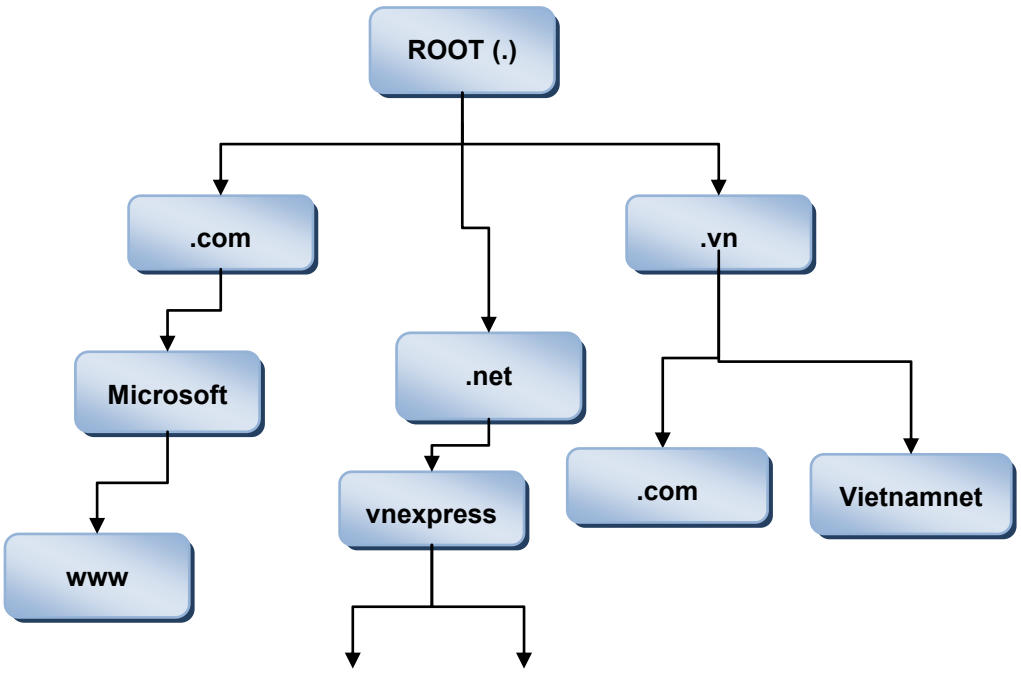
+	Bits 0 - 15	16 - 31
0	Source Port	Destination Port
32	Length	Checksum
64	Data	

Hình : UDP header

Một vài dịch vụ chạy trên UDP: Phân giải tên miền (DNS:53),RSTP,MMS…

2.6 Hệ thống phân giải tên miền

Trên Internet, mỗi máy tính muốn trao đổi dữ liệu với nhau đều phải biết địa chỉ IP. Với người dùng, việc nhớ 32 bit địa chỉ IPv4 hoặc 128 bit địa chỉ IPv6 là rất khó khăn do vậy người ta xây dựng hệ thống phân cấp, đặt tên cho các máy tính trên mạng để dễ nhớ: hệ thống phân giải tên miền (Domain Name System).



Hình 3: Phân cấp hệ thống phân giải tên miền

Tên miền được phân cấp và quản lý bởi INTERNIC. Cấp cao nhất là root, sau ngay sau đó là tên miền cấp 1, cấp 2, cấp 3…

Cấp	Cấp 4	Cấp 3	Cấp 2	Cấp 1
Tên miền	www.	hut.	edu.	vn

Trên Internet sẽ có các máy chủ riêng, chuyên thực hiện chức năng phân giải tên miền sang địa chỉ IP và ngược lại. Thông thường tổ chức được cấp một tên miền cấp 1 sẽ duy trì cơ sở dữ liệu tên miền cấp 2 trực tiếp, tổ chức cấp 2 lại duy trì tên miền cấp 3 trực tiếp…

Một máy tính muốn truy vấn địa chỉ IP của tên miền nào đó sẽ hỏi trực tiếp máy chủ phân giải tên miền mà nó nằm trong, máy chủ này nếu không trả lời được sẽ hỏi đến máy chủ cấp cao hơn, cấp cao hơn không trả lời được lại hỏi lên cấp cao nữa…

Dịch vụ phân giải tên miền chạy trên giao thức UDP, cổng 53.

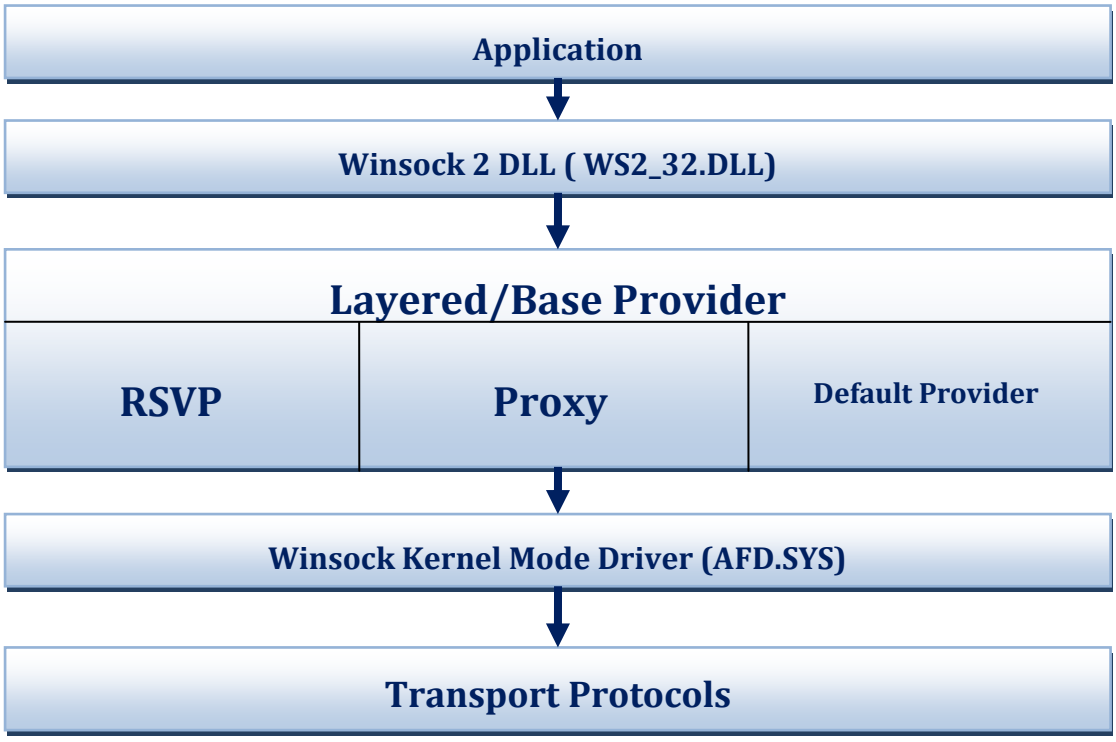
Chương 3. Winsock

Bài giảng số 3

- **Thời lượng: 3 tiết.**
- **Tóm tắt nội dung :**
  - **Kiến trúc WinSock.**
  - **Đặc tính của WinSock.**
    - **Hướng thông điệp**
    - **Hướng dòng**
    - **Hướng kết nối và không kết nối**
    - **Multicast**
    - **Chất lượng dịch vụ**

3.1 Kiến trúc

Winsock là bộ thư viện liên kết động đi kèm với họ hệ điều hành Windows của Microsoft. Winsock cung cấp các API để nhà phát triển có thể xây dựng các ứng dụng mạng đơn giản, hiệu năng cao. Winsock có vài phiên bản, bắt đầu từ phiên bản 1.0 được đưa ra năm 1992, cho đến nay là Winsock 2 được tích hợp vào tất cả các hệ điều hành mới của Microsoft. Kiến trúc Winsock gồm nhiều tầng, nhưng tầng trên cùng, giao tiếp trực tiếp với ứng dụng là thư viện WS2_32.DLL.



Hình 4: Kiến trúc Winsock

Ứng dụng sử dụng Winsock bằng cách liên kết và triệu gọi các hàm trong thư viện WS2_32.DLL. Thư viện thực hiện kiểm tra tính hợp lệ của các tham số, xác định giao thức tầng dưới thích hợp và chuyển lời gọi xuống thư viện giao thức bên dưới(Provider). Có thể có nhiều provider, winsock sẽ lựa chọn provider thích hợp. Các provider cung cấp các giao thức tầng mạng khác nhau như TCP/IP, IPX/SPX, AppleTalk, NetBIOS…

Các provider sau khi nhận dữ liệu từ tầng trên, xử lý và chuyển tiếp xuống tầng dưới, đó là driver ở chạy ở mức kernel của hệ điều hành (AFD.SYS). Driver này chịu trách nhiệm quản lý kết nối, bộ đệm vàn các tài nguyên liên quan đến socket, đồng thời giao tiếp với driver ở mức thấp hơn, driver điều khiển thiết bị phần cứng.

Ở mức thấp nhất là các Transport Protocols, hay còn gọi là các driver điều khiển thiết bị. Các driver này giao tiếp với tầng trên (AFD.SYS) thông quaTDI (Transport Driver Interface). TDI là giao diện chung của Microsoft, nhằm cung cấp một giao tiếp trong suốt, không phụ thuộc vào thiết bị. Các hãng phát triển phần cứng chỉ việc xây dựng trình điều khiển của mình tuân theo TDI, và nó sẽ phối hợp nhịp nhàng với hạ tầng mạng bên trên. Thiết kế này giải phóng Microsoft khỏi việc xây dựng những driver cụ thể và đảm bảo tính tương thích cho hệ điều hành.

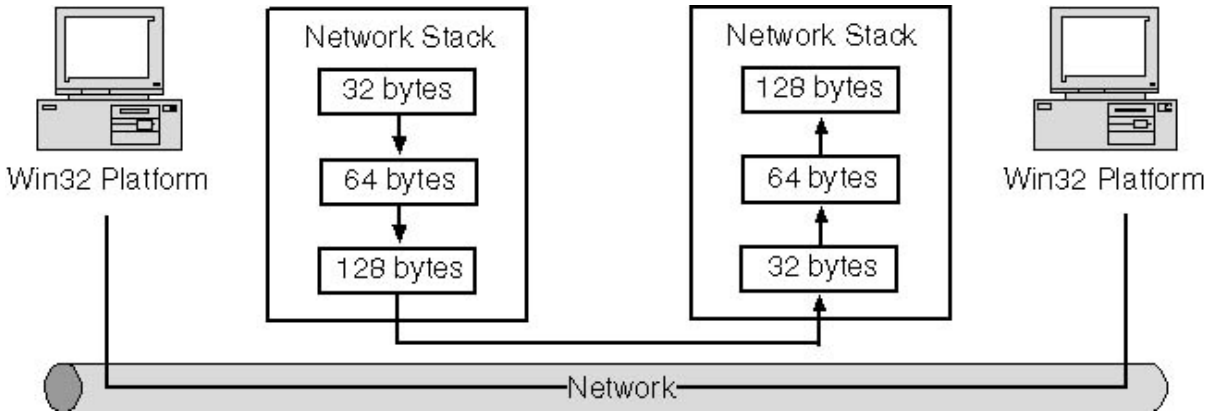
Việc lập trình ứng dụng mạng dựa trên Winsock sẽ chủ yếu thao tác với một đối tượng cơ bản SOCKET. SOCKET trừu tượng hóa tất cả các loại giao thức. Hai ứng dụng muốn trao đổi dữ liệu với nhau , mỗi bên phải tạo một socket, và đường dây ảo nối giữa hai SOCKET sẽ là kênh truyền dữ liệu. Tưởng tượng mỗi socket là một cái phích cắm, hai thiết bị muốn truyền thông tin thì cần có một dây dẫn hai đầu, mỗi đầu là một phích cắm cắm vào hai thiết bị đó.

3.2 Đặc tính

Winsock hỗ trợ nhiều giao thức mạng, thí dụ UDP, IP, TCP, IPX, Infared… Mỗi giao thức có những đặc tính riêng và việc sử dụng SOCKET cũng khác nhau với mỗi giao thức.

3.2.1 Giao thức hướng thông điệp

Giao thức được gọi là hướng thông điệp (Message-Oriented) nếu thông tin được truyền đi dưới dạng một thông điệp riêng lẻ. Một bên yêu cầu dữ liệu, nó sẽ chỉ nhận được một thông điệp đáp trả tương ứng với yêu cầu đã gửi đi. Thí dụ, một máy tính gửi 3 thông điệp, kích thước lần lượt là 32, 64, 128 byte. Bên nhận dù có nhận đủ cả 3 thông điệp vào bộ đệm hệ thống, nhưng ứng dụng muốn lấy thông điệp ra, nó phải thực hiện ba lần gọi hàm, và thứ tự nhận được mỗi lần là 32 ,64,128 byte. Phương pháp này bảo toàn biên của các thông điệp.

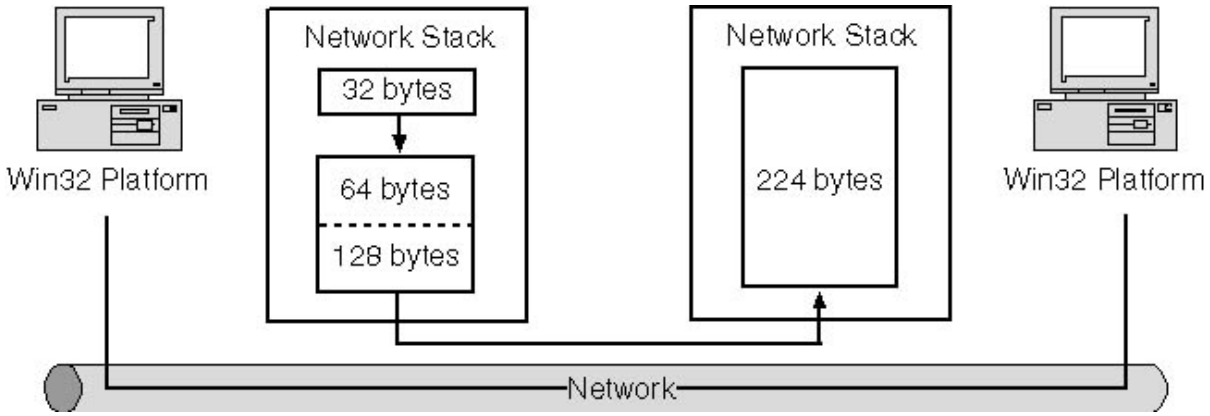


Hình 5. Giao thức hướng thông điệp

Giao thức hướng thông điệp thích hợp với các ứng dụng tổ chức truyền dữ liệu theo cấu trúc. Thí dụ, game chơi cờ trực tuyến, mỗi bên gửi một thông điệp chứa thông tin về nước đi của mình hoặc nhận thông điệp về nước đi của đối phương.

3.2.2 Giao thức hướng dòng

Giao thức không duy trì biên giữa các thông điệp được gọi là giao thức hướng dòng. Bên gửi và bên nhận truyền dữ liệu theo dòng, một cách liên tục mà không quan tâm đến biên giữa các lần truyền. Thí dụ, bên gửi gửi ba gói tin kích thước lần lượt là 32,64,128 byte, nhưng bên nhận nhận được một gói tin kích thước 224 byte là tổ hợp của ba gói tin trên.



Hình 6. Giao thức hướng dòng

3.2.3.Giao thức giả dòng

Giao thức giả dòng là những giao thức mà bên gửi chia dòng dữ liệu gửi đi thành các gói (thông điệp), bên nhận nhận các gói và ghép lại thành một dòng. TCP và UDP là hai giao thức giả dòng rất thông dụng hiện nay.

3.2.4 Giao thức hướng kết nối và không kết nối

Winsock hỗ trợ các giao thức hướng kết nối và không kết nối. Giao thức hướng kết nối nghĩa là đường truyền được thành lập giữa hai bên truyền nhận

trước khi dữ liệu thực sự được gửi đi, điều này đảm bảo có đường đi giữa hai bên, và đảm bảo hai bên cùng ở trạng thái hoạt động, sẵn sàng truyền dữ liệu. Tuy nhiên việc thành lập kết nối giữa hai bên sẽ làm tăng đáng kể dữ liệu phát sinh. Phần lớn các giao thức hướng kết nối đều cung cấp cơ chế kiểm soát lỗi, kiểm soát trật tự gói tin và kiểm soát mất mát, dư thừa, do đó tăng thêm tài nguyên tính toán. Ngược lại, giao thức không kết nối không cần thiết lập đường truyền, không cần đảm bảo bên nhận sẽ sẵn sàng nhận, nhận đúng, nhận đủ dữ liệu. Giao thức không kết nối cũng giống như việc gửi thư. Người gửi thư không biết người nhận có mong đợi nhận thư, cũng như khi nào nhận được hay bưu điện có thể chuyển được bức thư đến tay người nhận hay không.

Trong bộ giao thức TCP/IP, TCP là giao thức hướng kết nối còn UDP là giao thức không kết nối.

3.2.5 Tính tin cậy và đúng trật tự

Những đặc tính có lẽ quan trọng nhất khi lựa chọn một giao thức đó là tính tin cậy và đúng trật tự. Tính tin cậy trong một giao thức thể hiện ở việc nó sẽ đảm bảo chính xác từng byte được gửi mỗi bên, những giao thức không tin cậy sẽ không đảm bảo tính chất này.

Giao thức đúng trật tự đảm bảo chính xác trật tự dữ liệu giữa bên gửi và bên nhận. Byte nào gửi trước sẽ được nhận trước, byte gửi sau sẽ được nhận sau.

Giao thức hướng kết nối thường đảm bảo tính tin cậy và trật tự của dữ liệu, tuy nhiên chi phí xử lý sẽ tăng cao. Ngược lại, giao thức không kết nối thường không đảm bảo hai tính chất này, nhưng bù lại tốc độ và tính đáp ứng được đảm bảo, những loại ứng dụng thời gian thực và chấp nhận sai sót có thể sử dụng giao thức loại này.

3.2.6 Quá trình đóng kết nối

Việc thực hiện đóng kết nối chỉ xảy ra trong các giao thức hướng kết nối. Trong trường hợp của TCP, bên A muốn hủy phiên truyền, bên A sẽ gửi một đoạn tin với cờ FIN, bên B nhận được cờ FIN liên gửi đoạn tin trả lại A với cờ ACK để báo đã nhận được, lúc này A sẽ không thể gửi tin, nhưng vẫn có thể nhận tin, cho đến khi B gửi đoạn tin có cờ FIN, khi đó kết nối đã được đóng hoàn toàn.

3.2.7 Quảng bá dữ liệu

Winsock cũng hỗ trợ khả năng quảng bá dữ liệu của các giao thức. Với cơ chế này, một máy trạm gửi thông điệp tới tất cả các máy trạm khác trên LAN, giao thức không kết nối sẽ được sử dụng để truyền tin, hạn chế của phương pháp này là mỗi máy tính trong mạng sẽ mất thêm chi phí xử lý thông điệp dù muốn hay không muốn.

3.2.8 Multicast

Multicast là cơ chế gửi dữ liệu đến một hoặc nhiều máy trong mạng (không phải tất cả) thông qua một quá trình gọi là tham gia nhóm multicast. Thí dụ, với giao thức IP, các máy tính muốn nhận dữ liệu sẽ tham gia vào một nhóm multicast, bộ lọc sẽ được thực hiện trên phần cứng của card điều hợp mạng để chỉ xử lý dữ liệu liên quan đến nhóm multicast đó. Dữ liệu sau đó sẽ được đẩy ngược lên các tầng trên và chuyển cho ứng dụng thích hợp.

3.2.9 Chất lượng dịch vụ (QoS)

Chất lượng dịch vụ là cơ chế cho phép ứng dụng yêu cầu một băng thông dành riêng để sử dụng. Thí dụ, dịch vụ truyền hình thời gian thực, để ứng dụng bên nhận nhận được hình ảnh rõ ràng, liên tục thì bên gửi phải đáp ứng một vài tiêu chí về thời gian truyền và tốc độ truyền. QoS cho phép dành riêng một phần bằng thông trên mạng cho mục đích này, do đó dữ liệu truyền đi sẽ nhanh và đáp ứng kịp việc hiển thị ở bên nhận.

Bài giảng số 4

- **Thời lượng: 3 tiết.**
- **Tóm tắt nội dung :**
  - **Lập trình ứng dụng bằng WinSock.**
  - **Chuẩn bị môi trường.**
  - **Khởi tạo WinSock.**
  - **Thiết lập địa chỉ và cổng máy đích.**
  - **Sử dụng dịch vụ phân giải tên miền.**
  - **Tạo socket.**
  - **Truyền dữ liệu bằng giao thức TCP**
    - **Phần server.**
    - **Phần client.**

3.3 Lập trình Winsock

Phần này sẽ tập trung vào các thao tác cơ bản liên quan đến Winsock và TCP/IP, bao gồm khởi tạo, xây dựng TCP server, TCP client…

3.3.1 Môi trường

Môi trường cần thiết để xây dựng ứng dụng mạng winsock cần là:

- Hệ điều hành: Các hệ điều hành Win32 của Microsoft, bao gồm Windows 95/98/2000/Me/XP/2003/Vista/7. Giáo trình này sử dụng hệ điều hành Windows XP.
- Ngôn ngữ lập trình: Bất kỳ ngôn ngữ lập trình nào hỗ trợ việc gọi thư viện liên kết động đều có thể sử dụng để lập trình. Giáo trình này sử dụng ngôn ngữ C/C++.
- Thư viện: Winsock 2 bao gồm thư viện liên kết động WS2_32.DLL, tệp tiêu đề WINSOCK2.H, tệp thư viện WS2_32.LIB.
- Hướng dẫn: Thư viện trực tuyến MSDN

3.3.2 Khởi tạo Winsock

Mọi ứng dụng muốn sử dụng Winsock phải khởi tạo thư viện, Hàm WSAStartup sẽ làm nhiệm vụ khởi tạo đó.

```
int WSAStartup(
    WORD wVersionRequested,
    LPWSADATA lpWSAData
);
```

Trong đó wVersionRequested là phiên bản thư viện Winsock muốn nạp, BYTE thấp chứa số hiệu phiên bản chính, BYTE cao chứa phần lẻ. Macro MAKEWORD(x,y) sử dụng để tạo ra WORD cần thiết, với x là byte thấp, y là

byte cao. Như vậy có thể truyền wVersionRequested giá trị MAKEWORD(2,2) để khởi tạo phiên bản Winsock 2.2.

Tham số lpWSAData là con trỏ tới cấu trúc WSAData, Winsock sẽ điền thông tin về phiên bản vào trong cấu trúc này.

```
typedef struct WSAData
{
    WORD        wVersion;
    WORD        wHighVersion;
    char        szDescription[WSADESCRIPTION_LEN + 1];
    char        szSystemStatus[WSASYS_STATUS_LEN + 1];
    unsigned short iMaxSockets;
    unsigned short iMaxUdpDg;
    char FAR *    lpVendorInfo;
} WSADATA, * LPWSADATA;
```

Ý nghĩa của các trường có thể tra trong thư viện MSDN. Các phiên bản Winsock được cung cấp kèm với hệ điều hành như sau:

Nền tảng	Phiên bản Winsock
Windows 95	1.1 (2.2)
Windows 98	2.2
Windows Me	2.2
Windows NT 4.0	2.2
Windows 2000	2.2
Windows XP	2.2
Windows CE	1.1

Nếu việc khởi tạo thành công, hàm trả về giá trị 0, còn không trả về mã lỗi. Việc sử dụng phiên bản thư viện cao hơn phiên bản hệ điều hành hỗ trợ sẽ dẫn đến lỗi và trường wVersion sẽ trả về phiên bản cao nhất hệ điều hành đó hỗ trợ. Trong hầu hết các trường hợp, người ta thường khởi tạo phiên bản cao nhất mà hệ điều hành đó hỗ trợ. Dưới đây là đoạn mã khởi tạo thư viện Winsock hoàn chỉnh.

```
WSAData     wsaData;
WORD        wVersion = MAKEWORD(2,2);
```

```
if (WSAStartup(wVersion,&wsaData))
{
        printf("Version not supported");
}
```

Khi ứng dụng đã sử dụng xong Winsock, nó có thể giải phóng Winsock bằng lệnh WSACleanup. Lệnh này sẽ giải phóng mọi tài nguyên Winsock sử dụng, hủy các lệnh vào ra còn dang dở. Nguyên mẫu hàm như sau

```
int WSACleanup(void);
```

Nếu việc giải phóng thành công, hàm trả về 0, còn không thì là SOCKET_ERROR. Ứng dụng có thể lấy chi tiết lỗi bằng hàm WSAGetLastError().

Lưu ý: Hầu hết các hàm Winsock đều trả về 0 cho một thao tác thành công và SOCKET_ERROR cho thao tác thất bại. Để biết chi tiết nguyên nhân thất bại, ứng dụng có thể gọi hàm WSAGetLastError() ngay sau đó. Hàm sẽ trả về mã lỗi cụ thể mô tả nguyên nhân gây lỗi.

```
int WSAGetLastError (void);
```

3.3.3 Xác định địa chỉ

a.Xác định địa chỉ và cổng máy đích

Hai ứng dụng muốn truyền dữ liệu với nhau, trước hết phải biết địa chỉ của nhau, Winsock cung cấp một cấu trúc để người lập trình điền các thông tin địa chỉ đối tác trước khi thực hiện truyền nhận dữ liệu. Để đơn giản, ở đây sẽ chỉ sử dụng giao thức IPv4, IPv4 cũng là giao thức thông dụng nhất được sử dụng hiện nay. Cấu trúc địa chỉ của Winsock có dạng như sau:

```
struct sockaddr_in
{
   short          sin_family;
   u_short        sin_port;
   struct in_addr  sin_addr;
   char           sin_zero[8];
};
```

Trong đó:

- *sin_family* sẽ là *AF_INET*, giá trị này báo cho Winsock biết ứng dụng sẽ sử dụng họ địa chỉ IP.
- *sin_port* xác định cổng của giao thức TCP (UDP) sẽ kết nối đến.
- *sin_addr* là cấu trúc in_addr, thực chất là số nguyên 4 byte chứa địa chỉ IP của máy đích. Địa chỉ IP thường được biểu diễn dưới dạng a.b.c.d, mỗi byte trong *sin_addr* sẽ tương ứng với một giá trị trong chuỗi địa chỉ IP.
- *sin_zero* không có ý nghĩa ở đây, mục đích là làm cho cấu trúc sockaddr_in cùng kích thước với SOCKADDR.

Hàm *inet_addr* sẽ hữu ích trong việc chuyển đổi một xâu địa chỉ IP sang dạng số nguyên 32-bit.

```
unsigned long inet_addr(
   const char FAR *cp
);
```

Trong đó cp là xâu ký tự chứa địa chỉ IP dạng thập phân, đầu ra là số nguyên 32-bit dưới dạng đầu to (big-endian).

Hàm *inet_ntoa* làm nhiệm vụ chuyển đổi một địa chỉ IP sang dạng xâu.

```
char FAR *inet_ntoa(
                     struct in_addr in
);
```

Các bộ vi xử lý khác nhau xử lý số nguyên theo hai kiểu đầu nhỏ và đầu to tùy thuộc vào thứ tự sắp xếp các byte có trọng số lớn đến bé, tương ứng với hai

kiểu này. Bộ vi xử lý họ x86 của Intel lưu trữ số nguyên theo kiểu đầu nhỏ (hay còn gọi là host byte order), tuy nhiên Internet xử lý thông tin theo kiểu đầu to (network byte order). Do vậy trước khi truyền tham số phải chuyển đổi dữ liệu sang dạng đầu to. Winsock cung cấp các hàm sau phục vụ cho việc chuyển đổi.

Chuyển đổi từ host-byte order (little-endian => big-endian):

```
u_long htonl(u_long hostlong);        // Chuyển đổi 4 byte từ little-endian=>big-endian
int WSAHtonl(                         // Chuyển đổi 4 byte từ little-endian=>big-endian
   SOCKET s,
   u_long hostlong,
   u_long FAR * lpnetlong
);
u_short htons(u_short hostshort);     // Chuyển đổi 2 byte từ little-endian=>big-endian
int WSAHtons(                         // Chuyển đổi 2 byte từ little-endian=>big-endian
   SOCKET s,
   u_short hostshort,
   u_short FAR * lpnetshort
);
```

Chuyển đổi từ network-byte order (big-endian => little-endian):

```
u_long ntohl(u_long netlong);         // Chuyển 4 byte từ big-endian=>little-endian
int WSANtohl(                         // Chuyển 4 byte từ big-endian=>little-endian
   SOCKET s,
   u_long netlong,
   u_long FAR * lphostlong
);
u_short ntohs(u_short netshort);      // Chuyển 2 byte từ big-endian=>little-endian
int WSANtohs(                         // Chuyển 2 byte từ big-endian=>little-endian
   SOCKET s,
   u_short netshort,
   u_short FAR * lphostshort
);
```

Sau đây là đoạn chương trình khởi tạo cấu trúc địa chỉ cho Winsock:

```
SOCKADDR_IN InternetAddr;     // Khai báo cấu trúc địa chỉ
INT nPortId = 5150;           // Khai báo cổng
```

```
InternetAddr.sin_family = AF_INET;// Họ địa chỉ Internet
//Chuyển xâu địa chỉ 136.149.3.29 sang số 4 byte dang network-byte order và
//gán cho trường sin_addr
InternetAddr.sin_addr.s_addr = inet_addr("136.149.3.29");
//Chuyển đổi cổng sang dạng network-byte order và gán cho trường sin_port
InternetAddr.sin_port = htons(nPortId);
```

b.Phân giải tên miền

Trên thực tế, người ta thường không nhớ đến một máy tính thông qua địa chỉ IP, mà thông qua tên miền. Việc kết nối đến một ứng dụng hay dịch vụ trên một máy chủ từ xa sẽ được thực hiện thông qua tên miền, thí dụ: www.google.com, www.hut.edu.vn. Winsock cung cấp các hàm API hỗ trợ ứng dụng thực hiện phân giải tên miền sang địa chỉ IP để kết nối.

Winsock 1.1 sử dụng các hàm *gethostbyname* và *inet_addr*, tuy nhiên các hàm này chỉ hỗ trợ chuyển đổi sang IPv4. Từ phiên bản 2 trở đi, Winsock cung cấp hai hàm mới *getnameinfo* và *getaddrinfo*, hỗ trợ cả IPv4 và IPv6. Để sử dụng các hàm này cần *include* thư viện WSPIAPI.H và WS2TCPIP.H. Chương trình sẽ chạy được trên tất cả các hệ điều hành họ Windows hỗ trợ Winsock 2. Nguyên mẫu của hàm getaddrinfo như sau:

```
int getaddrinfo(
                    const char FAR *nodename,
                    const char FAR *servname,
                    const struct addrinfo FAR *hints,
                    struct addrinfo FAR *FAR *res
);
```

Trong đó

- nodename: là tên miền hoặc địa chỉ cần phân giải, thí dụ www.google.com.vn.
- servname: xâu chứa số nguyên hoặc chuỗi mô tả dịch vụ. Thí dụ “ftp” và “21” là tương đương nhau.
- hint: con trỏ đến cấu trúc addrinfo chứa gợi ý cho hàm thực hiện.
- res: con trỏ đến đầu cấu trúc danh sách liên kết kiểu addrinfo, chứa danh sách các kết quả phân giải được.
- Hàm trả về 0 nếu thành công, còn không là mã lỗi.

Cấu trúc *addrinfo* được định nghĩa như sau:

```
struct addrinfo {
```

```
                int             ai_flags;
                int             ai_family;
                int             ai_socktype;
                int             ai_protocol;
                size_t          ai_addrlen;
                char            *ai_canonname;
                struct sockaddr *ai_addr;
                struct addrinfo *ai_next;
};
```

Cấu trúc *hint* phải được xóa trước khi truyền, chỉ 4 trường đầu liên quan mới cần được thiết lập.

Trong đó:

- *ai_flags* nhận một trong các giá trị sau: AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST. Với AI_CANONNAME ám chỉ *nodename* là tên miền, AI_NUMERICHOST ám chỉ nodename là địa chỉ IP, thí dụ "192.168.1.2".
- *ai_family* nhận một trong các giá trị sau: AF_INET, AF_INET6, AF_UNSPEC, tương ứng với việc sẽ nhận về địa chỉ IPv4, IPv6 hoặc cả hai.
- *ai_socktype* xác định kiểu socket, thường là SOCK_DGRAM cho UDP và SOCK_STREAM cho TCP.
- *ai_protocol* xác định kiểu giao thức, thường là IPPROTO_TCP.

Nếu không cung cấp cấu trúc *hint*, hàm sẽ hoạt động như với *ai_family* là AF_UNSPEC.

Nếu thực hiện phân giải thành công, kết quả được trả về qua con trỏ *res*. Nếu có nhiều hơn một kết quả (Một tên miền có thể tương ứng với nhiều địa chỉ IP và ngược lại), res sẽ là danh sách liên kết chứa tất cả các kết quả, trường *ai_next* sẽ trỏ đến kết quả tiếp theo, trường ai_addr chứa kết quả phân giải được, trường *ai_addrlen* chứa chiều dài của ai_addr.

Đoạn mã sau sẽ thực hiện phân giải địa chỉ tên miền *www.hut.edu.vn*

```
struct addrinfo             hints,
                            *result;
int                         rc;
memset(&hints, 0, sizeof(hints));
hints.ai_flags = AI_CANONNAME;
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
```

```
rc = getaddrinfo("www.hut.edu.vn", "http", &hints, &result);
if (rc != 0) {
                // không phân giải được
}
freeaddrinfo(result);
```

Danh sách addrinfo là danh sách do hệ thống cấp phát, do vậy phải giải phóng danh sách sau khi dung. Winsock cung cấp hàm *freeaddrinfo* thực hiện điều đó.

3.3.4 Tạo socket

Sau khi đã có đầy đủ thông tin về máy đích, viêc đầu tiên cần làm để kết nối đến là tạo một socket, tức là tạo một cổng kết nối ảo từ máy cục bộ nối tới máy đích. Hàm để tạo socket có nguyên mẫu như sau:

```
SOCKET socket (
   int af,
   int type,
   int protocol
);
```

Tham số đầu tiên, *af* (Address Family) xác định họ socket, trong giáo trình này sẽ chỉ sử dụng IPv4 do đó giá trị của af là AF_INET.

Tham số thứ hai, *type* xác định kiểu socket. Với IPv4, thường có hai loại SOCK_STREAM cho TCP/IP và SOCK_DGRAM cho UDP/IP.

Tham số cuối cùng xác định giao thức sử dụng cho tầng giao vận, với TCP sẽ là IPPROTO_TCP, với UDP là IPPROTO_UDP.

Thí dụ sau đây sẽ tạo một socket TCP trên nền IPv4.

```
SOCKET s;
s = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)
```

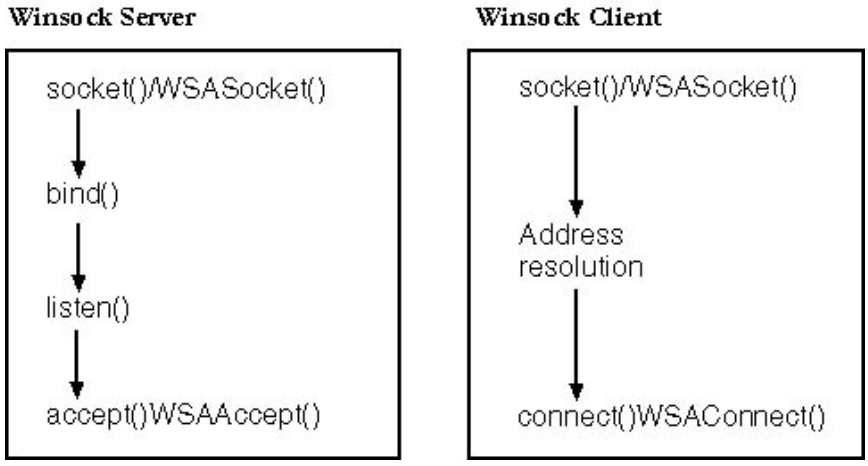
3.3.5 Truyền dữ liệu sử dụng giao thức (TCP)

Mô hình truyền dữ liệu cơ bản của TCP/IP là client-server, nghĩa là một bên sẽ đóng vài trò máy khách, một bên là máy chủ phục vụ. Việc xây dựng ứng dụng truyền dữ liệu qua môi trường mạng cũng phải gồm hai phần như vậy: client và server. Mỗi phần sẽ có cách xử lý khác nhau một chút.

Giao thức TCP cung cấp cơ chế truyền dữ liệu tin cậy, chính xác và đúng trật tự. Ứng dụng sử dụng TCP sẽ thiết lập một kênh truyền ảo giữa máy tính nguồn và đích. Khi kênh truyền đã được thiết lập, dữ liệu sẽ truyền giữa hai máy tính như hai dòng byte riêng biệt.

a.Phần server

Công việc của TCP server là liên tục lắng nghe và đáp ứng các yêu cầu kết nối của client từ một giao diện và cổng nào đó. Thí dụ một máy chủ web có địa chỉ IP là 202.191.56.69, ứng dụng Web server sẽ phải liên tục lắng nghe và chấp nhận các yêu cầu từ client thông qua giao diện 202.191.56.69:80. Công việc đầu tiên của server là tạo một socket thông qua hàm *socket* hoặc *WSASocket*. Tiếp theo *bind* socket vào một giao diện và cổng nào đó trên máy cục bộ, việc tạo socket cũng giống như chúng ta mua một phích cắm, việc *bind* giống như lựa chọn ổ cắm nào đó còn trống trong nhà để cắm vào. Việc tiếp theo sau khi bind là chuyển socket sang chế độ *listen* (đợi kết nối). Cuối cùng, khi có yêu cầu kết nối từ client, server phải chấp nhận kết nối thông qua hàm *accept* hoặc *WSAAccept.* Một server có thể chấp nhận kết nối từ nhiều client, mỗi lần chấp nhận thành công, một socket mới được tạo ở phía server và socket này chỉ sử dụng để truyền dữ liệu với client tương ứng.



Hình 7: Trình tự hoạt động của server và client

Bind

Việc tạo socket là như nhau giữa server và client, và đã được đề cập ở phần trước. Với server, việc tiếp theo sau khi tạo socket là *bind*. Nguyên mẫu hàm *bind* như sau:

```
int bind(
  SOCKET                 s,
  const struct sockaddr FAR*   name,
  int                    namelen
);
```

Trong đó *s* là một socket đã tạo trước đó, thực chất s là một số nguyên định danh tài nguyên socket mà Winsock sử dụng, *name* là con trỏ tới cấu trúc *sockaddr* chứa thông tin về giao diện và cổng server muốn bind, *namelen* chứa

chiều dài của cấu trúc sockaddr đó. Hàm trả về 0 nếu thành công, SOCKET_ERROR nếu thất bại, sử dụng WSAGetLastError() để lấy về mã lỗi, lỗi thông thường là bind và một cổng đã được bind trước đó rồi.

Lưu ý cấu trúc sockaddr là cấu trúc chung sử dụng cho nhiều giao thức, các cấu trúc khác như sockaddr_in đều có kích thước bằng sockaddr và có các trường đặc trưng cho giao thức internet, vì vậy khi làm việc với TCP/IP, có thể sử dụng sockaddr_in thay thế sockaddr. Thí dụ dưới đây minh họa việc sử dụng *bind* với server chạy ở cổng 8888.

```
SOCKET                  s;
SOCKADDR_IN             tcpaddr;
int                     port = 8888;

s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);// Tao socket

tcpaddr.sin_family = AF_INET;// Socket kieu IPv4
tcpaddr.sin_port = htons(port);  // Chuyen port tu host-byte order => net-byte order
tcpaddr.sin_addr.s_addr = htonl(INADDR_ANY); //Su dung bat ky giao dien nao

bind(s, (SOCKADDR *)&tcpaddr, sizeof(tcpaddr)); // Bind socket
```

Listen

Khi socket đã được *bind* thành công, nó có thể sẵn sàng chuyển sang trạng thái *listening* để lắng nghe kết nối từ client. Hàm *listen* sẽ thực hiện điều đó.

```
int listen(
  SOCKET s,
  int    backlog
);
```

Tham số s chỉ định một socket hợp lệ và đã được *bind*. Tham số backlog xác định chiều dài hàng đợi với server, tham số này quan trọng trong trường hợp có nhiều kết nói đến server cùng một lúc nhưng server chưa xử lý kịp và sẽ được đưa vào hàng đợi. Nếu hàng đợi đầy, các kết nối khác từ client sẽ bị hệ thống từ chối. Thông thường, chiều dài hàng đợi bị hạn chế bởi driver, nếu thiết lập backlog giá trị không hợp lệ, hệ thống sẽ chọn giá trị hợp lệ gần nhất.

```
listen(s,100);
```

Chấp nhận kết nối

Server sau khi *listen* đã có thể sẵn sàng chấp nhận kết nối từ các client khác, Winsock cung cấp các hàm thực hiện việc đó accept, AcceptEx, WSAAccept. Nguyên mẫu các hàm này như sau:

```
SOCKET accept(
   SOCKET s,
   struct sockaddr FAR* addr,
   int FAR* addrlen
);
```

Hàm *accept* nhận đầu vào là một socket hợp lệ, đang ở trạng thái *listening*, đầu ra là thông tin về client kết nối đến qua con trỏ *addr* có cấu trúc SOCKADDR_IN và chiều dài của cấu trúc qua biến *addrlen*. Nếu s là blocking socket, accept sẽ chặn luồng gọi hàm cho đến khi có client kết nối đến. Kết quả trả về của hàm là một socket tương ứng với client được chấp nhận , socket này đã sẵn sàng cho việc gửi nhận dữ liệu. Nếu s là non-blocking socket (socket bất đồng bộ), và tại thời điểm gọi hàm, chưa có client nào kết nối đến, *accept* sẽ trả về WSAEWOULDBLOCK. Nếu s không phải là socket hợp lệ, *accept* sẽ trả về SOCKET_ERROR. Các hàm AcceptEx và WSAAccept sẽ được mô tả cụ thể hơn ở phần sau. Dưới đây là đoạn chương trình khởi tạo và chấp nhận kết nối của server.

```
#include <winsock2.h>                //Thu vien Winsock
void main(void)
{
  WSADATA            wsaData;
  SOCKET             ListeningSocket;
  SOCKET             NewConnection;
  SOCKADDR_IN        ServerAddr;
  SOCKADDR_IN        ClientAddr;
  int                ClientAddrLen;
  int              Port = 8888;

  // Khoi tao Winsock 2.2
  WSAStartup(MAKEWORD(2,2), &wsaData);

   // Tao socket lang nghe ket noi tu client.
```

```
    ListeningSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    // Khoi tao cau truc SOCKADDR_IN cua server
    // doi ket noi o cong 8888

    ServerAddr.sin_family = AF_INET;
    ServerAddr.sin_port = htons(Port);
    ServerAddr.sin_addr.s_addr = htonl(INADDR_ANY);

    // Bind socket cua server.

    bind(ListeningSocket, (SOCKADDR *)&ServerAddr,
    sizeof(ServerAddr));

  // Chuyen sang trang thai doi ket noi
    listen(ListeningSocket, 5);

  // Chap nhan ket noi moi.
    NewConnection = accept(ListeningSocket, (SOCKADDR *)
                    &ClientAddr,&ClientAddrLen);
  // Sau khi chap nhan ket noi, server co the tiep tuc chap nhan them cac ket noi khac,
  //  hoac gui nhan du lieu voi cac client thong qua cac socket duoc accept voi client

  // Dong socket
    closesocket(NewConnection);
    closesocket(ListeningSocket);

  // Giai phong Winsock
    WSACleanup();
}
```

b.Phần client

Sử dụng socket trong client tương đối đơn giản hơn server. Các công việc client cần thực hiện:

- Khởi tạo một socket

- Điền thông tin về server sẽ kết nối đến vào cấu trúc SOCKADDR_IN, trong đó đặc biệt quan trọng là địa chỉ server và cổng.
- Thực hiện *connect* hoặc *WSAConnect* để tạo kết nối đến server và truyền nhận dữ liệu.

Việc khởi tạo socket và điền thông tin cấu trúc SOCKADDR_IN đã nói ở phần trên. Nguyên mẫu hàm *connect* như sau:

```
int connect(
   SOCKET s,
   const struct sockaddr FAR* name,
   int namelen
);
```

Với s là socket được tạo bởi hàm *socket*, name là con trỏ trỏ tới cấu trúc SOCKADDR_IN chứa thông tin về server, namelen là chiều dài cấu trúc SOCKADDR_IN.

Nếu không có server nào chạy ở máy tính kết nối đến, hay không có tiến trình nào đợi ở cổng mà client muốn kết nối đến, *connect* sẽ trả về lỗi WSAECONNREFUSED. Nếu có lỗi trên đường truyền hoặc máy tính kết nối đến không tồn tại, hàm sẽ trả về WSAETIMEDOUT.

Đoạn chương trình sau sẽ thực hiện kết nối đến server có địa chỉ www.hut.edu.vn và cổng 8888.

```
#include <winsock2.h>

void main(void)
{
  WSADATA              wsaData;
  SOCKET          s;
  SOCKADDR_IN          ServerAddr;
  int                  Port = 8888;


  // Khoi tao Winsock 2.2

  WSAStartup(MAKEWORD(2,2), &wsaData);

  // Tao socket client.
```

```
    s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

  // Khoi tao cau truc SOCKADDR_IN co dia chi server la 202.191.56.69 va cong 8888

    ServerAddr.sin_family = AF_INET;
    ServerAddr.sin_port = htons(Port);
    ServerAddr.sin_addr.s_addr = inet_addr("202.191.56.69");

  // Ket noi den server thong qua socket s.

    connect(s, (SOCKADDR *) &ServerAddr, sizeof(ServerAddr));

  // Bat dau gui nhan du lieu

  // Ket thuc gui nhan du lieu
  //  Dong socket
    closesocket(s);

  // Giai phong Winsock
  // WSACleanup.

    WSACleanup();
}
```

c. Gửi nhận dữ liệu giữa client và server

Việc gửi và nhận dữ liệu giữa server và client diễn ra sau khi kết nối đã được thiết lập, tức là client đã *connect* thành công, server đã *accept* thành công. Lúc này cặp socket mà client sử dụng để connect (s), và socket mà server accept (NewClient) sẽ dùng để gửi và nhận dữ liệu giữa hai bên. Vai trò client và server là như nhau trong cặp socket này. Để gửi dữ liệu qua socket, Winsock cung cấp hai hàm *send* và *WSASend*. Hàm *recv* và *WSARecv* sẽ nhận dữ liệu từ socket. Dữ liệu trong Winsock là chuỗi byte liên tiếp, Winsock không phân biệt ký tự, số hay xâu. Nếu việc gửi và nhận thành công, hàm trả về số byte gửi hay nhận được, còn không sẽ là SOCKET_ERROR (-1). Để lấy thông tin mã lỗi, chương trình có thể gọi *WSAGetLastError* ngay sau đó. Các lỗi thường gặp là WSAECONNABORTED và WSAECONNRESET, các lỗi này xảy ra khi một trong hai bên đóng kết nối, hoặc do lỗi đường truyền. Một lỗi khác cũng thường gặp là WSAEWOULDBLOCK, thực chất đây không phải là lỗi, chỉ xuất hiện trên các

socket non-blocking (bất đồng bộ) có ý nghĩa là socket không thể gửi hay nhận dữ liệu ngay tại thời điểm đó.

Nguyên mẫu hàm *send* gửi dữ liệu như sau:

```
int send(
   SOCKET s,
   const char FAR * buf,
   int len,
   int flags
);
```

Tham số đầu tiên, *s* là socket đã được kết nối, và dữ liệu sẽ gửi đi trên socket này. Tham số thứ hai *buf* là con trỏ đến bộ đệm dữ liệu cần gửi. Tham số *len* là chiều dài bộ đệm. Tham số cuối cùng flags là cờ chỉ định cách thức gửi dữ liệu, flags có thể là 0, MSG_DONTROUTE, MSG_OOB hoặc kết hợp của các cờ trên theo phép OR. MSG_DONTROUTE nghĩa là báo cho tầng giao vận không định tuyến gói tin này, MSG_OOB báo cho tầng giao vận biết đây là gói tin Out-of-Band. Thông thường *flags* nhận giá trị 0.

Nếu gửi thành công, hàm sẽ trả về số byte gửi được. Nếu thất bại hàm sẽ trả về SOCKET_ERROR, mã lỗi cụ thể có được khi gọi WSAGetLastError có thể là WSAECONNABORTED, WSAECONNRESET, WSAETIMEDOUT.

Minh họa lệnh *send* trên socket đã kết nối *s* của client.

```
char szHello[]=”Hello Network Programming”;
send(s,szHello,strlen(szHello),0);
```

Từ phiên bản 2, Winsock cung cấp thêm hàm WSASend để gửi dữ liệu:

```
int WSASend(
   SOCKET s,
   LPWSABUF lpBuffers,
   DWORD dwBufferCount,
   LPDWORD lpNumberOfBytesSent,
   DWORD dwFlags,
   LPWSAOVERLAPPED lpOverlapped,
   LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Trong đó s cũng là socket đã được kết nối, LPWSABUF là mảng các cấu trúc WSABUF mô tả các bộ đệm chứa dữ liệu cần gửi, dwBufferCount là số lượng bộ

đệm có trong mảng lpBuffers, LPDWORD là con trỏ sẽ chứa số byte gửi được, dwFlags tương đương với flags trong hàm send. Hai tham số cuối cùng sử dụng trong phương pháp vào ra bất đồng bộ, sẽ được mô tả cụ thể hơn ở phần sau.

Giả sử s là socket đã được kết nối, thí dụ sau đây sẽ dùng WSASend để gửi chuỗi "Hello Network Programming" đến server:

```
char        szHello[]="Hello Network Programming";
WSABUF      buffs[10];
DWORD       dwBytesSent;
buffs[0].len = strlen(szHello);
buffs[0].buf = szHello;
WSASend(s,buffs,1,&dwBytesSent,0,0,0);
```

Nhận dữ liệu từ socket được thực hiện thông qua hai hàm *recv* và *WSARecv* (Winsock 2).

Nguyên mẫu hàm *recv*

```
int recv(
  SOCKET s,
  char FAR* buf,
  int len,
  int flags
);
```

Trong đó *s* là socket đã kết nối, *buf* là con trỏ đến bộ đệm chứa dữ liệu nhận được, *len* là chiều dài bộ đệm hay số byte muốn nhận, *flags* quy định cách thức nhận dữ liệu. Các giá trị có thể có của *flags* là 0, MSG_PEEK, MSG_OOB hoặc kết hợp của các cờ trên. Nếu *flags* là 0, không hành động đặc biệt nào được thực hiện. Nếu *flags* là MSG_OOB, Winsock sẽ nhận về dữ liệu Out-of-Band. Nếu *flags* là MSG_PEEK, Winsock sẽ copy dữ liệu ra *buf* những vẫn giữ nguyên giữ liệu trong bộ đệm hệ thống. Nếu hàm thực hiện thành công, giá trị trả về là số byte nhận được, còn không giá trị trả về là SOCKET_ERROR.

Thí dụ sử dụng lệnh recv để nhận dữ liệu từ socket s

```
char buf[100];
int len = 0;
len = recv(s,buf,100,0);
```

Winsock 2 cung cấp thêm hàm WSARecv, hàm này hỗ trợ vào ra bất đồng bộ và gửi nhận từng phần datagram.

```
int WSARecv(
```

```
  SOCKET s,
  LPWSABUF lpBuffers,
  DWORD dwBufferCount,
  LPDWORD lpNumberOfBytesRecvd,
  LPDWORD lpFlags,
  LPWSAOVERLAPPED lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Với s là socket đã kết nối, lpBuffers là mảng các bộ đệm kiểu WSABUF, dwBufferCount là số lượng bộ đệm, lpNumberOfBytesRecvd là con trỏ lưu số byte nhận được, lpFlags là con trỏ chứa cờ quy định hoạt động của hàm, lpOverlapped và lpCompletionRoutine sử dụng trong vào ra bất đồng bộ sẽ đề cập trong phần sau. Tham số lpFlags vừa là giá trị vào, vừa là giá trị ra và có thể nhận các giá trị sau 0, MSG_PEEK, MSG_OOB, MSG_PARTIAL hoặc kết hợp của các giá trị thông qua phép OR. Ba giá trị đầu tương tự như hàm recv, MSG_PARTIAL chỉ sử dụng với các giao thức hướng thông điệp. Trong trường hợp nó là tham số vào, hệ thống sẽ gửi trả dữ liệu ngay khi thông điệp vừa nhận được mà không quan tâm đã nhận đủ thông điệp hay chưa, còn trong trường hợp nó là tham số ra, hệ thống sẽ thiết lập cờ này nếu bộ đệm không đủ để nhận toàn bộ một thông điệp, và cờ này báo hiệu rằng dữ liệu chỉ là một phần của thông điệp. Giao thức TCP sẽ không sử dụng cờ này.

Thí dụ nhận 100 byte từ socket *s* vào bộ đệm *buf*

```
char charbuf[100];
WSABUF wsaBuf;
DWORD dwBytesRcvd = 0;
wsaBuf.buf = charbuf;
wsa.len = 100;
WSARecv(s,&wsaBuf,1,&dwBytesRcvd,0,0);
```

Lưu ý: Mặc dù trong nguyên mẫu hàm send, recv, WSASend, WSARecv có quy định số lượng byte ứng dụng muốn gửi hoặc nhận, nhưng nếu số lượng này lớn hơn kích thước cửa sổ TCP thì hệ thống cũng không thể gửi toàn bộ dữ liệu qua một lần gọi hàm. Chương trình nên thực hiện kiểm tra số lượng byte thực sự đã được gửi (hoặc nhận) để lặp lại việc gửi cho đến khi hoàn tất.

Sau khi quá trình gửi nhận hoàn tất, client hoặc server có thể đóng kết nối bằng lệnh *shutdown*:

```
int shutdown(
```

```
  SOCKET s,
  int how
);
```

Với *s* là socket cần đóng, *how* chỉ ra cách thức đóng, có thể là SD_RECEIVE, SD_SEND hoặc SD_BOTH. Hàm *shutdown* đóng kết nối một cách tường minh và hệ thống đảm bảo ứng dụng nhận được dữ liệu còn thừa trước khi đóng hoàn toàn kết nối.

Sau khi đóng kết nối, ứng dụng có thể gọi hàm *closesocket* để giải phóng mọi tài nguyên liên quan đến socket đó:

```
int closesocket (SOCKET s);
```

Hàm này sẽ giải phóng mọi tài nguyên sử dụng, loại bỏ mọi dữ liệu đang xử lý dở dang và đóng kết nối.

Chương trình client sau sẽ gửi thông điệp “Hello Network Programming” tới server ở địa chỉ www.hut.edu.vn và cổng 8888, địa chỉ server có thể thay đổi cho phù hợp với thực tế.

```
#include <winsock2.h>
#include <ws2tcpip.h>
void main(void)
{
  WSADATA              wsaData;
  SOCKET               s;
  SOCKADDR_IN          ServerAddr;
  int                  Port = 8888;
  char                 szHello[] = “Hello Network Programming”;
  addrinfo             hints,*result; // Lưu địa chỉ IP của server

  // Khởi tạo Winsock 2.2

  WSAStartup(MAKEWORD(2,2), &wsaData);

  // Tạo socket client.

    s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  // Khởi tạo cấu trúc hints
    memset(&hints,0,sizeof(addrinfo));
   hints.ai_family = AF_INET;
```

```
   hints.ai_flags = AI_CANONNAME;
   hints.ai_protocol = IPPROTO_TCP;
   hints.ai_socktype = SOCK_STREAM;
  // Truy vấn địa chỉ IP của server
   rc = getaddrinfo("www.hut.edu.vn","8888",&hints,&result);
  if (!rc) // Thất bại
     return 0;
    // Kết nối đến server thông qua socket s.

    connect(s, (SOCKADDR *)result->ai_addr, sizeof(SOCKADDR));

  // Gửi chuỗi hello
   send(s,szHello,strlen(szHello),0);
  // Đóng kết nối
   shutdown(s,SD_BOTH);
   closesocket(s);

  // Giai phong Winsock
  // WSACleanup.

    WSACleanup();
}
```

Chương trình sau làm nhiệm vụ server, đợi ở cổng 8888 và hiển thị nội dung client gửi tới

```
#include <winsock2.h>                  // Thư viện Winsock
void main(void)
{
  WSADATA           wsaData;
  SOCKET            ListeningSocket;
  SOCKET            NewConnection;
  SOCKADDR_IN       ServerAddr;
  SOCKADDR_IN       ClientAddr;
  int               Port = 8888;
  char                    buf[100];
  int                     len;
  int                   ClientAddrLen = sizeof(ClientAddr);
```

```
// Khởi tạo Winsock 2.2
WSAStartup(MAKEWORD(2,2), &wsaData);

 // Tạo socket lắng nghe.

  ListeningSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

  // Khoi tao cau truc SOCKADDR_IN cua server
  // doi ket noi o cong 8888

  ServerAddr.sin_family = AF_INET;
  ServerAddr.sin_port = htons(Port);
  ServerAddr.sin_addr.s_addr = htonl(INADDR_ANY);

  // Bind socket cua server.

  bind(ListeningSocket, (SOCKADDR *)&ServerAddr,
  sizeof(ServerAddr));

// Chuyen sang trang thai doi ket noi
  listen(ListeningSocket, 5);

// Chap nhan ket noi moi.
  NewConnection = accept(ListeningSocket, (SOCKADDR *)
                &ClientAddr,&ClientAddrLen);
// Nhận dữ liệu từ client
  len = recv(NewConnection,buf,100,0);
  buf[len] = 0;
 // Hiển thị ra màn hình
  printf("Dữ liệu nhận dc từ client: %s",buf);

// Đóng kết nối
  shutdown(NewConnection,SD_BOTH);
  closesocket(NewConnection);
  closesocket(ListeningSocket);
```

```
  // Giai phong Winsock
    WSACleanup();
}
```

Bài giảng số 5

- **Thời lượng: 3 tiết.**
- **Tóm tắt nội dung :**
  - **Truyền nhận dữ liệu bằng giao thức UDP.**
    - **Bên gửi.**
    - **Bên nhận**
  - **Một vài hàm vào ra khác.**

3.3.6 Truyền dữ liệu sử dụng giao thức UDP

Việc gửi nhận dữ liệu sử dụng giao thức không kết nối khá khác với hướng kết nối. Trong bộ TCP/IP, giao thức hỗ trợ hình thức truyền dữ liệu này là UDP. UDP không đảm bảo tính tin cậy của dữ liệu, có thể gửi đến nhiều đích và nhận nhận từ nhiều nguồn. Việc truyền nhận cũng không cần phải thiết lập kết nối trước, và không có cơ chế báo nhận.

a.Bên nhận

Các công việc cần thực hiện để nhận dữ liệu tương đối đơn giản. Đầu tiên, tạo một socket bằng hàm *socket* hoặc *WSASocket*. Tiếp theo *bind* socket vừa tạo vào một giao diện nào đó. Cuối cùng hàm *recvfrom* sẽ nhận dữ liệu datagram từ bất kỳ một máy tính nào trong mạng mà không cần phải thực hiện *listen* hay accept trước đó.

```
int recvfrom(
  SOCKET s,
  char FAR* buf,
  int len,
  int flags,
  struct sockaddr FAR* from,
  int FAR* fromlen
);
```

Hàm *recvfrom* nhận dữ liệu từ socket nào đó, bốn tham số đầu tương tự như với hàm recv, hai tham số cuối chứa thông tin về máy nguồn gửi datagram đó.

```
int WSARecvFrom(
  SOCKET s,
  LPWSABUF lpBuffers,
  DWORD dwBufferCount,
  LPDWORD lpNumberOfBytesRecvd,
  LPDWORD lpFlags,
  struct sockaddr FAR * lpFrom,
```

```
  LPINT lpFromlen,
  LPWSAOVERLAPPED lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Hàm này tương tự như với WSARecv, nhưng có thêm tham số *lpFrom* và *lpFromlen* ghi nhận máy tính nguồn gửi datagram đến.

Đoạn chương trình sau sẽ nhận dữ liệu datagram từ cổng 8888 và hiển thị ra màn hình.

```
SOCKET              s;
SOCKADDR_IN         addr,source;
int                 len = sizeof(source);
// Tạo socket
receiver = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
// Khởi tạo địa chỉ và cổng
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(8888); // Đợi UDP datagram ở cổng 8888

// Bind socket vào địa chỉ cục bộ và cổng 8888
bind(receiver,(sockaddr*)&addr,sizeof(SOCKADDR_IN));

// Lặp đợi gói tin
while (1)
{
        // Nhận dữ liệu từ mạng
        datalen = recvfrom(ListeningSocket,buf,100,0,(sockaddr*)&source,&len);
        // Kiểm tra chiều dài
        if (datalen>0)
        {
                buf[datalen]=0;
                printf("Data:%s",buf); // Hiển thị ra màn hình
        }
}
```

Ngoài việc nhận dữ liệu bằng hàm *recvfrom* và *WSARecvFrom*. Ứng dụng có thể gọi hàm recv để nhận dữ liệu từ socket. Tuy nhiên trước đó ứng dụng phải *connect* tới địa chỉ của máy gửi, hàm *connect* sẽ không thực hiện việc kết nối

đến máy gửi, thay vì đó nó sẽ báo cho Winsock biết ứng dụng chỉ mong muốn nhận dữ liệu từ máy tính có địa chỉ được chỉ định trong cấu trúc SOCKADDR_IN mà connect sử dụng, chứ không phải bất kỳ máy tính nào khác trên mạng.

b.Bên gửi

Bên gửi có thể gửi dữ liệu datagram thông qua *sendto* hoặc *WSASendTo*.

```
int sendto(
  SOCKET s,
  const char FAR * buf,
  int len,
  int flags,
  const struct sockaddr FAR * to,
  int tolen
);
```

Bốn tham số đầu có ý nghĩa tương tự như hàm *send*. Tuy nhiên socket s không cần phải *bind* hay *connect* tới đâu cả. Hai tham số cuối chứa thông tin về địa chỉ máy tính đích sẽ nhận dữ liệu. Phiên bản 2.0 của hàm này có dạng sau.

```
int WSASendTo(
  SOCKET s,
  LPWSABUF lpBuffers,
  DWORD dwBufferCount,
  LPDWORD lpNumberOfBytesSent,
  DWORD dwFlags,
  const struct sockaddr FAR * lpTo,
  int iToLen,
  LPWSAOVERLAPPED lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Ý nghĩa các tham số cũng tương tự WSASent, ngoại trừ việc socket không cần phải *connect* trước đó. Hàm cần cung cấp thêm thông tin về địa chỉ máy đích thông qua cặp tham số lpTo và iToLen.

Đoạn chương trình sau sẽ gửi một xâu “Hello Network Program” dưới dạng một datagram tới địa chỉ IP 202.191.56.69 và cổng 8888.

```
char buf[]=”Hello Network Programming”;
SOCKET sender;
SOCKADDR_IN receiverAddr;
// Tạo socket để gửi tin
```

```
SendingSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
 // Điền địa chỉ đích
ReceiverAddr.sin_family = AF_INET;
ReceiverAddr.sin_port = htons(8888);
ReceiverAddr.sin_addr.s_addr = inet_addr("202.191.56.69");
// Thực hiện gửi tin
sendto(sender, buf, strlen(buf), 0,
        (SOCKADDR *)&receiverAddr, sizeof(recieverAddr));
```

Tương tự như recvfrom, việc gửi datagram cũng có thể thực hiện bằng hàm *send* hoặc *WSASend* nếu ứng dụng đã gọi *connect* hoặc *WSAConnect* trên socket trước đó. Địa chỉ đích gửi đi sẽ luôn là địa chỉ được truyền trong hàm *connect* hoặc *WSAConnect*. Việc gọi *connect* trên một datagram socket chỉ có ý nghĩa báo cho Winsock địa chỉ đích cho mọi lời gọi *send* hoặc *WSASend* trên socket sau đó.

Lưu ý: Trong giao thức không kết nối, không cần thực hiện lời gọi shutdown để đóng kết nối, khi ứng dụng hoàn tất việc gửi nhận dữ liệu, tất cả những gì cần làm là gọi hàm closesocket để giải phóng socket.

3.3.7 Một vài hàm khác

Winsock cung cấp khá nhiều hàm API, dưới đây liệt kê một vài hàm thông dụng khi lập trình.

```
int getpeername(
   SOCKET s,
   struct sockaddr FAR* name,
   int FAR* namelen
);
```

Hàm này sử dụng để lấy thông tin về địa chỉ máy tính đầu kia của socket. Tham số đầu tiên là socket đã được kết nối, hai tham số ra sau chứa thông tin về địa chỉ socket của máy kia.

```
int getsockname(
   SOCKET s,
   struct sockaddr FAR* name,
   int FAR* namelen
);
```

Hàm này sử dụng để lấy địa chỉ cục bộ của socket. Tham số đầu tiên là socket đã được kết nối, hai tham số sau là đầu ra của hàm, chứa thông tin về địa chỉ cục bộ của socket.

Bài giảng số 6

- **Thời lượng: 3 tiết**
- **Tóm tắt nội dung:**
  - **Các phương pháp vào ra**
    - **Các chế độ hoạt động**
      - **Đồng bộ**
      - **Bất đồng bộ**
    - **Các mô hình vào ra**
      - **Mô hình blocking**
      - **Mô hình select**
      - **Mô hình WSAAsyncSelect**
      - **Mô hình WSAEventSelect**
      - **Mô hình Overlapped**

3.4 Các phương pháp vào ra

3.4.1 Các chế độ hoạt động của Winsock

Winsock có hai chế độ hoạt động blocking (đồng bộ) và non-blocking (bất đồng bộ). Hai chế độ này khác nhau ở cách trở về từ các hàm. Ở chế độ đồng bộ, các hàm gửi nhận dữ liệu như *send*, *recv* sẽ chỉ trở về nơi gọi khi thao tác gửi nhận hoàn tất, và như vậy nó sẽ chặn (block) hoạt động của luồng (thread) có lời gọi hàm đó đến khi hoàn tất. Nếu việc triệu gọi các thao tác vào ra đồng bộ diễn ra trong luồng xử lý giao diện, thì giao diện của chương trình sẽ đáp ứng rất chậm chạp và mang lại cảm giác không thoải mái. Ngược lại, ở chế độ bất đồng bộ, các hàm Winsock sẽ trở về ngày lập tức bất kể thao tác gửi nhận dữ liệu có hoàn tất hay chưa hoàn tất.

a.Chế độ đồng bộ

Phần lớn các ứng dụng Winsock hoạt động theo vòng lặp nhận dữ liệu – xử lý, tức là ứng dụng sẽ nhận một ít dữ liệu, thực hiện xử lý trên đó và lại nhận dữ liệu - xử lý…

```
SOCKET          sock;
char            buff[256];
int             done = 0, nBytes;
...
while(!done) // Chừng nào chưa kết thúc
{
  nBytes = recv(sock, buff, 65);    // Nhận dữ liệu
  if (nBytes == SOCKET_ERROR) // Nếu có lỗi
   {
```

```
        printf("recv failed with error %d\n",
        WSAGetLastError());
        return;
    }
    DoComputationOnData(buff); // Thực hiện tính toán khác
}
...
```

Bắt đầu mỗi vòng lặp, ứng dụng đợi dữ liệu nhận về và thực hiện xử lý, nếu không có dữ liệu về việc tính toán và các xử lý khác không thể thực hiện tiếp. Nếu việc nhận dữ liệu thực hiện trong trong luồng xử lý giao diện (GUI Thread), thì ứng dụng sẽ không thể đáp ứng được các sự kiện từ người dùng. Giải pháp thường được đưa ra ở đây là chuyển việc nhận dữ liệu vào một luồng riêng và sử dụng cơ chế đồng bộ để xử lý dữ liệu.

Giả sử ứng dụng cần nhận dữ liệu từ mạng, dữ liệu là các thông điệp, mỗi thông điệp có kích thước *NUM_BYTES_REQUIRED*. Đoạn chương trình sau chia việc nhận và xử lý thành hai luồng *ReadThread* và *ProcessThread* đồng bộ với nhau thông qua đoạn găng *data* và sự kiện *hEvent* đã được tạo trước đó. Luồng *ReadThread* sẽ lặp liên tục để nhận đủ *NUM_BYTES_REQUIRED* và bộ đệm buff. Sau đó thông qua sự kiện hEvent báo cho luồng *ProcessThread* biết dữ liệu sẵn sàng để xử lý.

```
#define MAX_BUFFER_SIZE    4096 // Kích thước tối đa của bộ đệm
// Khai báo đoạn găng
CRITICAL_SECTION data;
// Khai báo biến sự kiện
HANDLE        hEvent;

SOCKET        sock;
TCHAR         buff[MAX_BUFFER_SIZE];
int           done=0;



// Tạo và kết nối socket
...
// Luồng nhận dữ liệu
void ReadThread(void)
{
    int nTotal = 0,      // Số byte nhận được tổng cộng
```

```
    nRead = 0,          // Số byte nhận được mỗi lần
    nLeft = 0,          // Số byte còn lại của thông điệp
    nBytes = 0;

  while (!done)    // Chừng nào chưa kết thúc
  {
    // Khởi đầu mỗi vòng lặp
    nTotal = 0; // Tổng số byte đã nhận được trong mỗi lần lặp
    nLeft = NUM_BYTES_REQUIRED;   // Số byte còn lại của thông điệp

    // Lặp việc nhập dữ liệu cho đến khi nhận đủ NUM_BYTE_REQUIRED
    while (nTotal != NUM_BYTES_REQUIRED)
    {
      EnterCriticalSection(&data); // Vào đoạn găng
      nRead = recv(sock, &(buff[MAX_BUFFER_SIZE - nBytes]),
        nLeft, 0);
      if (nRead == -1)
      {
        printf("error\n");
        ExitThread();
      }
      nTotal += nRead;
      nLeft -= nRead;

      nBytes += nRead;
      LeaveCriticalSection(&data); // Ra đoạn găng
    }
    SetEvent(hEvent); // Báo hiệu luồng ProcessThread dữ liệu trong buff đã sẵn sàng
  }
}

// Luồng xử lý dữ liệu
void ProcessThread(void)
{
  // Đợi sự kiện nhận đủ một thông điệp từ luồng ReadThread
  WaitForSingleObject(hEvent);
```

```
  // Vào đoạn găng
  EnterCriticalSection(&data);
  DoSomeComputationOnData(buff);
  //Lấy dữ liệu ra khỏi bộ đệm
  nBytes -= NUM_BYTES_REQUIRED;
  // Ra khỏi đoạn găng
  LeaveCriticalSection(&data);
}
```

Việc xử lý như trên áp dụng với nhiều socket một lúc là khá phức tạp khi mỗi kết nối cần đến hai luồng, nếu tính cả việc gửi dữ liệu đi, thì cần đến ba luồng và hiệu năng hệ thống chưa được tối ưu.

b. Chế độ bất đồng bộ

Ở chế độ bất đồng bộ, các hàm gửi nhận sẽ trở về ngay lập tức bất kể việc gửi và nhận đã hoàn tất hay chưa hoàn tất. Các socket mặc định khi được tạo sẽ hoạt động ở chế độ đồng bộ. Đoạn lệnh sau sẽ chuyển socket sang chế độ bất đồng bộ.

```
SOCKET      s;
unsigned long ul = 1;
int         nRet;

s = socket(AF_INET, SOCK_STREAM, 0);
nRet = ioctlsocket(s, FIONBIO, (unsigned long *) &ul);
if (nRet == SOCKET_ERROR)
{
  // Thất bại
}
```

Ở chế độ bất đồng bộ, các hàm gửi nhận dữ liệu của WinSock sẽ trở về ngay lập tức với mã lỗi là *WSAWOULDBLOCK*. Đây thực chất không phải lỗi, chỉ là giá trị báo hiệu rằng WinSock chưa có đủ thời gian để gửi dữ liệu. Người lập trình sẽ phải có cơ chế kiểm tra khác để biết khi nào việc gửi nhận dữ liệu đã hoàn tất. Các hàm sau đây sẽ trả về lỗi WSAWOULDBLOCK nếu WinSock hoạt động ở chế độ bất đồng bộ.

Tên hàm	Mô tả
WSAAccept,accept	Ứng dụng chưa nhận được yêu cầu kết nối nào.
closesocket	Kết nối chưa thực sự được đóng.
WSAConnect,connect	Kết nối đã được khởi tạo nhưng chưa hoàn tất.

WSARecv, recv, WSARecvFrom, recvfrom	Chưa nhận được dữ liệu nào.
WSASend, send, WSASendTo, and sendto	Dữ liệu chưa thể gửi đi ngay lập tức .

3.4.2 Các mô hình vào ra

Mô hình vào ra là các cơ chế để ứng dụng trao đổi dữ liệu với WinSock. Có tất cả 6 mô hình vào ra: blocking, select, WSAAsyncSelect, WSAEventSelect, Overlapped và completion port. Phần này sẽ trình bày năm mô hình vào ra đầu tiên.

a.Mô hình blocking

Đây là mô hình đơn giản nhất. Do việc sử dụng các hàm gửi nhận dữ liệu sẽ chặn luồng hiện tại, nên mô hình này sử dụng hai luồng độc lập cho việc gửi và nhận dữ liệu. Ưu điểm duy nhất của mô hình này là đơn giản và dễ phát triển, hạn chế là không thể mở rộng ra nhiều kết nối bởi việc đó đồng nghĩa với việc phải tạo nhiều luồng trên hệ thống và sẽ không hiệu quả về mặt tài nguyên.

b.Mô hình select

Đây là mô hình được sử dụng rộng rãi. Thông qua việc sử dụng hàm *select* ứng dụng có thể biết được nó có thể gửi dữ liệu đi không, hay có dữ liệu nhận được đang chờ hay không. Hàm *select* sẽ chặn luồng hiện tại cho đến khi một trong các điều kiện mà nó *select* được thỏa mãn. Nguyên mẫu của hàm select như sau

```
int select(
   int nfds,
   fd_set FAR * readfds,
   fd_set FAR * writefds,
   fd_set FAR * exceptfds,
   const struct timeval FAR * timeout
);
```

Tham số đầu tiên nfds bị bỏ qua trong WinSock và nó chỉ được đưa vào với mục đích tương thích với các chương trình viết trên nền socket của Berkeley. Các tham số còn lại readfds, writefds, exceptfds là tập các socket mà hàm select sẽ kiểm tra. Kiểu dữ liệu fd_set là tập các socket. Thí dụ, hàm select sẽ thành công nếu một trong các socket của tập readfds thỏa mãn điều kiện:

- Có dữ liệu nhận được.
- Kết nối bị đóng, reset hoặc hủy.
- Nếu hàm *listen* đã được gọi trước đó và hàm *accept* thành công.

Tương tự như vậy, select sẽ thành công nếu một trong các socket của tập writefds thỏa mãn điều kiện:

- Dữ liệu có thể gửi đi.
- Nếu hàm *connect* thành công trên socket bất đồng bộ.

Cuối cùng select sẽ thành công nếu một trong các socket của tập exceptfds thỏa mãn điều kiện

- Nếu hàm *connect* thất bại trên một socket bất đồng bộ.
- Nếu nhận được dữ liệu OOB.

Các tập readfds, writefds, exceptfds có thể NULL, nhưng không thể cả ba cùng NULL.

Tham số cuối cùng *timeout* sẽ quyết định hàm *select* sẽ đợi bao lâu trước khi trở về. Cấu trúc của timeval như sau:

```
struct timeval
{
   long tv_sec;
   long tv_usec;
};
```

Trong đó tv_sec tính theo đơn vị giây và tv_usect là mili giây. Nếu giá trị timeout là {0,0} có nghĩa là hàm *select* sẽ chỉ thăm dò trạng thái của các socket và trở về ngay lập tức.

Nếu select thành công, nó sẽ trả về số lượng socket có sự kiện tương ứng. Nếu sau khoảng thời gian timeout mà không có sự kiện nào xảy ra, select sẽ trả về 0. Nếu vì bất kỳ lý do gì khác, select sẽ trả về SOCKET_ERROR.

Trước khi có thể sử dụng hàm select, cần phải khởi tạo các cấu trúc fd_set. WinSock cung cấp các MACRO sau để thao tác với cấu trúc này:

- FD_ZERO(*set): Khởi tạo một tập rỗng.
- FD_CLR(s,*set): Xóa bỏ socket s ra khỏi tập s.
- FD_ISSET(s,*set): Kiểm tra xem socket s có được thiết lập hay không.
- FD_SET(s,*set): Thêm socket s vào tập s.

Đoạn chương trình sau sẽ dùng lệnh select để kiểm tra trạng thái của socket s

```
SOCKET  s;
fd_set  fdread;
int     ret;
// Khởi tạo socket s và tạo kết nối

// Thao tác vào ra trên socket s
while(TRUE)
{
   // Xóa tập fdread
```

```
  FD_ZERO(&fdread);
  // Thêm s vào tập fdread
  FD_SET(s, &fdread);
  if ((ret = select(0, &fdread, NULL, NULL, NULL))  // Đợi sự kiện trên socket s
    == SOCKET_ERROR)
  {
    // Error condition
  }

  if (ret > 0)
  {
    if (FD_ISSET(s, &fdread)) // Kiểm tra xem s có được thiết lập hay không
    {
      // Xử lý sự kiện nhận dữ liệu từ s
    }
  }
}
```

Ưu điểm lớn nhất của mô hình *select* là nó cho phép nhiều socket có thể thao tác trên cùng một luồng. Mặc định số socket tối đa trong một tập là 64, về lý thuyết có thể có đến 1024 socket trong một tập. Tuy nhiên việc cho quá nhiều socket vào một tập cũng ảnh hưởng đến hiệu năng khi phải duyệt qua tất cả các socket mỗi khi có một sự kiện xảy ra với một trong các socket nằm trong tập đó.

c.Mô hình WSAAsyncSelect

WinSock cung cấp cơ chế vào ra bất động bộ dựa trên thông điệp của Windows. Đây là cơ chế cho phép một ứng dụng GUI nhận được thông điệp mạng của WinSock. Để nhận được thông điệp, ứng dụng phải tạo ít nhất một cửa sổ. Việc lập trình giao diện để tạo cửa sổ trong Windows sẽ không được đề cập đến ở đây. Khi cửa sổ đã được tạo, ứng dụng sẽ gọi hàm WSAAsyncSelect để báo cho WinSock về cửa sổ sẽ nhận thông điệp.

```
int WSAAsyncSelect(
  SOCKET s,
  HWND hWnd,
  unsigned int wMsg,
  long lEvent
);
```

Trong đó:

- Tham số s mô tả socket sẽ được xử lý thông điệp.
- hWnd là handle của cửa sổ sẽ nhận thông điệp.

- wMsg mô tả thông điệp sẽ gửi đến cửa sổ để phân biệt với các thông điệp khác. Đây là tham số tùy chọn, ứng dụng thường chọn giá trị wMsg lớn hơn WM_USER.
- lEvent là mặt nạ mô tả các loại thông điệp mà ứng dụng sẽ được nhận. Thường là FD_READ, FD_WRITE, FD_ACCEPT, FD_CONNECT, FD_CLOSE.

Thí dụ:

```
WSAAsyncSelect(s, hwnd, WM_SOCKET, FD_CONNECT | FD_READ | FD_WRITE | FD_CLOSE);
```

Khi ứng dụng gọi hàm này, WinSock sẽ tự động chuyển socket tương ứng sang chế độ bất đồng bộ.

Khi cửa sổ nhận được thông điệp, hệ điều hành sẽ triệu gọi hàm WindowsProc (một loại hàm callback) tương ứng với cửa sổ đó. Nguyên mẫu của hàm như sau:

```
LRESULT CALLBACK WindowProc(
  HWND hWnd,
  UINT uMsg,
  WPARAM wParam,
  LPARAM lParam };
```

Trong đó hWnd là cửa sổ nhận được sự kiện, uMsg là thông điệp tương ứng với cửa sổ đó, ứng dụng phải đối chiếu uMsg với thông điệp đã thiết lập trong hàm WSAAsyncSelect để biết đó có phải là thông điệp mạng hay không. Tham số wParam chính là socket xảy ra sự kiện. Tham số cuối cùng lParam chứa hai thông tin, nửa thấp chứa mã của sự kiện, nửa cao chứa mã lỗi nếu có. Thí dụ việc xử lý sự kiện trong cửa sổ như sau:

```
BOOL CALLBACK WinProc(HWND hDlg,UINT wMsg,
  WPARAM wParam, LPARAM lParam)
{
  SOCKET Accept;

  switch(wMsg)
  {
    case WM_PAINT:
      // Xử lý sự kiện khác
      break;

    case WM_SOCKET:
      // Sự kiện WinSock
```

```
        // Kiểm tra có lỗi hay không
        if (WSAGETSELECTERROR(lParam))
        {
            // Đóng socket
           closesocket( (SOCKET) wParam);
           break;
        }

        // Xác định sự kiện
        switch(WSAGETSELECTEVENT(lParam))
        {
           case FD_ACCEPT:

              // Chấp nhận kết nối
              Accept = accept(wParam, NULL, NULL);
              ....
              break;
           case FD_READ:
              // Nhận dữ liệu từ socket wParam
              ...
              break;
           case FD_WRITE:
              // Gửi dữ liệu đến socket wParam
              break;
           case FD_CLOSE:
              // Đóng kết nối
              closesocket( (SOCKET)wParam);
              break;
        }
        break;
   }
   return TRUE;
}
```

Mô hình vào ra WSAAsyncSelect có nhiều ưu điểm, thí dụ có thể xử lý nhiều sự kiện đồng thời mà không quá mất công thiết lập, kiểm tra cấu trúc fd_set như mô hình select. Tuy nhiên nhược điểm là yêu cầu ứng dụng phải có cửa sổ. Những ứng dụng không cần cửa sổ (console, dịch vụ) sẽ không có cơ hội sử dụng mô hình vào ra này. Đồng thời, nếu trên một phạm vi lớn, một cửa sổ phải xử lý hàng ngàn sự kiện với các socket có thể không phải là giải pháp tối ưu.

d.Mô hình WSAEventSelect

Mô hình này sử dụng cơ chế đồng bộ theo sự kiện trên Windows. Mỗi socket sẽ có một biến đồng bộ sự kiện riêng. Biến sự kiện sẽ được khởi tạo bởi hàm WSACreateEvent có nguyên mẫu như sau:

```
WSAEVENT    WSACreateEvent(void);
```

Hàm này tạo ra một đối tượng sự kiện ở trạng thái chưa được báo hiệu (non-signaled) và thiết lập thủ công (manual reset). Sau khi tạo đối tượng sự kiện, ứng dụng sẽ sử dụng hàm WSAEventSelect để gắn đối tượng sự kiện với socket tương ứng.

```
int WSAEventSelect(
  SOCKET s,
  WSAEVENT hEventObject,
  long lNetworkEvents
);
```

Trong đó s là socket sẽ được xử lý sự kiện, hEventObject là đối tượng sẽ nhận sự kiện, lNetworkEvents là mặt nạ quy định các sự kiện sẽ được WinSock gửi đi. Đối tượng sự kiện có hai trạng thái là đã báo hiệu (signaled) và chưa báo hiệu (non-signed) và hai chế độ hoạt động là thiết lập thủ công (manual reset) và thiết lập tự động (auto reset). Đối tượng sự kiện được tạo ra ở chế độ thiết lập thủ công, nghĩa là mỗi khi sự kiện được báo hiệu (signaled), ứng dụng phải chuyển nó về chế độ chưa báo hiệu (non-signaled) thủ công thông qua hàm

```
BOOL WSAResetEvent(WSAEVENT hEvent);
```

Khi việc xử lý kết thúc, ứng dụng sẽ hủy đối tượng sự kiện bằng hàm

```
BOOL WSACloseEvent(WSAEVENT hEvent);
```

Khi đối tượng sự kiện đã được tạo và gắn vào socket cụ thể, ứng dụng sẽ sử dụng hàm WaitForMultipleEvents để đợi sự kiện trên các socket đó.

Nguyên mẫu hàm như sau

```
DWORD WSAWaitForMultipleEvents(
  DWORD cEvents,
  const WSAEVENT FAR * lphEvents,
  BOOL fWaitAll,
  DWORD dwTimeout,
  BOOL fAlertable
);
```

Các tham số:

- cEvents là số lượng biến sự kiện sẽ đợi, theo lý thuyết có thể có tối đa 64 đối tượng sự kiện.
- lphEvents là mảng các biến sự kiện
- fWaitAll quyết định sẽ đợt tất cả các biến sự kiện chuyển sang trạng thái đã báo hiệu hay chỉ cần một trong các biến sự kiện đã báo hiệu là đủ
- dwTimeout, thời gian tối đa tính bằng mili giây mà hàm sẽ đợi.
- fAlertable có thể tạm bỏ qua và nên thiết lập là FALSE.

Giá trị trả về của hàm là thứ tự của sự kiện đầu tiên chuyển sang trạng thái báo hiệu trong mảng các sự kiện lphEvents, và xác định bằng giá trị trả về của hàm trừ đi WSA_WAIT_EVENT_0:

```
Index = WSAWaitForMultipleEvents(...);
MyEvent = EventArray[Index - WSA_WAIT_EVENT_0];
```

Khi đã xác định được đối tượng sinh ra sự kiện, ứng dụng cần xác định mã của sự kiện cụ thể là gì. Hàm WSAEnumNetworkEvents sẽ thực hiện điều đó.

```
int WSAEnumNetworkEvents(
  SOCKET s,
  WSAEVENT hEventObject,
  LPWSANETWORKEVENTS lpNetworkEvents
);
```

Mã của sự kiện sẽ nằm trong mảng lpNetworkEvents có cấu trúc như sau:

```
typedef struct _WSANETWORKEVENTS
{
  long lNetworkEvents;
  int  iErrorCode[FD_MAX_EVENTS];
} WSANETWORKEVENTS, FAR * LPWSANETWORKEVENTS;
```

lNetworkEvents là mặt nạ chứa mã các sự kiện thí dụ FD_READ, FD_WRITE…và iErrorCode là mảng các mã lỗi tương ứng với sự kiện đó.

Đoạn mã sau đây sẽ minh họa việc sử dụng sự kiện trong server để quản lý nhiều kết nối một lúc.

```
SOCKET SocketArray [WSA_MAXIMUM_WAIT_EVENTS]; // Mảng các socket
// Mảng các đối tượng sự  kiện
WSAEVENT EventArray [WSA_MAXIMUM_WAIT_EVENTS],   NewEvent;
SOCKADDR_IN InternetAddr;
SOCKET Accept, Listen;
DWORD EventTotal = 0;
DWORD Index, i;
// Thiết lập socket server đợi kết nối ở cổng 8888
Listen = socket (PF_INET, SOCK_STREAM, 0);
```

```
InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(8888);

bind(Listen, (PSOCKADDR) &InternetAddr,
   sizeof(InternetAddr));

// Tạo đối tượng sự kiện đợi kết nối mới
NewEvent = WSACreateEvent();

// Gắn đối tượng sự kiện vào socket Listen
WSAEventSelect(Listen, NewEvent,
   FD_ACCEPT | FD_CLOSE);
// Chuyển sang chế độ đợi kết nối
listen(Listen, 5);

// Khởi phần tử đầu tiên cho mảng socket và mảng đối tượng sự kiện
SocketArray[EventTotal] = Listen;
EventArray[EventTotal] = NewEvent;
EventTotal++;

while(TRUE)
{
   // Đợi sự kiện mạng trên các socket
   Index = WSAWaitForMultipleEvents(EventTotal,
      EventArray, FALSE, WSA_INFINITE, FALSE);

   // Xác định chỉ số của socket gây ra sự kiện
   Index = Index - WSA_WAIT_EVENT_0;

   // Duyệt qua tất cả các socket
   for(i=Index; i < EventTotal ;i++
   {
   // Kiểm tra lại lần nữa với từng socket nhưng với timeout bằng 1000 ms
    Index = WSAWaitForMultipleEvents(1, &EventArray[i], TRUE, 1000,
     FALSE);
    if ((Index == WSA_WAIT_FAILED) || (Index == WSA_WAIT_TIMEOUT))
       continue;
    else
    {
       Index = i;
      // Xác định các sự kiện với socket thứ i
       WSAEnumNetworkEvents(
```

```
    SocketArray[Index],
    EventArray[Index],
   &NetworkEvents);
// Kiểm tra sự kiện FD_ACCEPT
if (NetworkEvents.lNetworkEvents & FD_ACCEPT)
{
    if (NetworkEvents.iErrorCode[FD_ACCEPT_BIT] != 0) // Nếu có lỗi
    {
        printf("FD_ACCEPT failed with error %d\n",
            NetworkEvents.iErrorCode[FD_ACCEPT_BIT]);
        break;
    }
    // Chấp nhận kết nối mới và lưu vào socket Accept
    Accept = accept(SocketArray[Index],NULL, NULL);
  // Nếu có quá nhiều kết nối => đóng socket
    if (EventTotal > WSA_MAXIMUM_WAIT_EVENTS)
    {
        printf("Too many connections");
        closesocket(Accept);
        break;
    }
     // Tạo sự kiện cho socket vừa tạo
    NewEvent = WSACreateEvent();
   // Gắn sự kiện vào socket vừa tạo
    WSAEventSelect(Accept, NewEvent,
        FD_READ | FD_WRITE | FD_CLOSE);
   // Lưu vào mảng sự kiện và mảng socket
    EventArray[EventTotal] = NewEvent;
    SocketArray[EventTotal] = Accept;
    EventTotal++;
    printf("Socket %d connected\n", Accept);
}
// Xử lý sự kiện FD_READ
if (NetworkEvents.lNetworkEvents & FD_READ)
{
    // Nếu có lỗi ?
    if (NetworkEvents.iErrorCode[FD_READ_BIT] != 0)
    {
        printf("FD_READ failed with error %d\n",
            NetworkEvents.iErrorCode[FD_READ_BIT]);
        break;
    }
```

```
        // Nhận dữ liệu từ socket
        recv(SocketArray[Index - WSA_WAIT_EVENT_0], buffer, sizeof(buffer), 0);
      }

      // Xử lý sự kiện FD_WRITE
      if (NetworkEvents.lNetworkEvents & FD_WRITE)
      {
        // Kiểm tra lỗi
        if (NetworkEvents.iErrorCode[FD_WRITE_BIT] != 0)
        {
            printf("FD_WRITE failed with error %d\n",
            NetworkEvents.iErrorCode[FD_WRITE_BIT]);
          break;
        }
        // Gửi dữ liệu nếu cần
        send(SocketArray[Index - WSA_WAIT_EVENT_0],
          buffer, sizeof(buffer), 0);
        }

        // Xử lý sự kiện đóng socket
        if (NetworkEvents.lNetworkEvents & FD_CLOSE)
        {
          if (NetworkEvents.iErrorCode[FD_CLOSE_BIT] != 0)
          {
            printf("FD_CLOSE failed with error %d\n",
              NetworkEvents.iErrorCode[FD_CLOSE_BIT]);
            break;
          }
          closesocket(SocketArray[Index]);
          // Loại bỏ đối tượng sự kiện và socket ra khỏi mảng EventArray và
SocketArray
          CompressArrays(EventArray, SocketArray, &EventTotal);
        }
      }
  }
}
```

Cơ chế hoạt động của mô hình *WASEventSelect* có phần tương tự so với *select* .Ưu điểm của mô hình này là không cần môi trường Windows để có thể nhận sự kiện. Tuy vậy có hạn chế là mỗi luồng chỉ hỗ trợ 64 socket cùng một lúc. Ứng dụng cần nhiều socket hơn phải tạo thêm luồng và do vậy nó cũng không thích hợp mới quy mô lớn.

e.Mô hình Overlapped

Đây là mô hình vào ra mạnh nhất. Mô hình này cho phép ứng dụng gửi một hoặc nhiều yêu cầu vào ra bất đồng bộ và xử lý các yêu cầu vào ra đã hoàn tất vào thời điểm sau đó. Mô hình này tương tự như cơ chế vào ra trên Windows thông qua các hàm ReadFile và WriteFile.

Để sử dụng mô hình vào ra này, socket phải được tạo bằng hàm *WSASocket* với cờ overlapped được bật.

```
s = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP,0,0, WSA_FLAG_OVERLAPPED).
```

Sau đó các hàm sau đây có thể sử dụng với mô hình vào ra này:

- WSASend
- WSASendTo
- WSARecv
- WSARecvFrom
- WSAIoctl
- WSARecvMsg
- AcceptEx
- ConnectEx
- TransmitFile
- TransmitPackets
- DisconnectEx
- WSANSPIoctl

Để sử dụng mô hình này, mỗi hàm vào ra nhận thêm một tham số là con trỏ tới cấu trúc WSAOVERLAPPED. Các hàm sử dụng cấu trúc này sẽ kết thúc ngay lập tức bất kể chế độ hoạt động của socket là đồng bộ hay bất đồng bộ. Cấu trúc này mô tả các thông tin cần thiết để hoàn thành thao tác vào ra. Có hai phương pháp để xử lý kết quả của thao tác: hoàn thành thông qua sự kiện hoặc thông qua chương trình con.

Xử lý hoàn thành thông qua sự kiện. Ứng dụng sẽ đợi kết quả vào ra thông qua đối tượng sự kiện trong cấu trúc WSAOVERLAPPED có khai báo như sau:

```
typedef struct WSAOVERLAPPED
{
  DWORD   Internal;
  DWORD   InternalHigh;
  DWORD   Offset;
```

```
  DWORD    OffsetHigh;
  WSAEVENT hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

Các trường Internal, InternalHigh, Offset, OffsetHigh được WinSock sử dụng nội bộ, ứng dụng không nên thao tác trực tiếp trên các trường này. Trường hEvent là đối tượng sự kiện sẽ nhận được thông báo khi tháo tác vào ra hoàn tất, ứng dụng sẽ tạo đối tượng sự kiện và điền vào cấu trúc này trước khi truyền vào các hàm vào ra. Khi thao tác vào ra hoàn tất, WinSock sẽ chuyển trạng thái của đối tượng sự kiện từ chưa báo hiệu sang trạng thái đã báo hiệu. Nhiệm vụ của ứng dụng là lấy về kết quả của thao tác vào ra thông qua hàm *WSAGetOverlappedResult.*

```
BOOL WSAGetOverlappedResult(
  SOCKET s,
  LPWSAOVERLAPPED lpOverlapped,
  LPDWORD lpcbTransfer,
  BOOL fWait,
  LPDWORD lpdwFlags
);
```

Trong đó

- s là socket tương ứng với sự kiện.
- lpOverlapped là con trỏ tới cấu trúc overlapped đã sử dụng khi bắt đầu thao tác vào ra.
- lpcbTransfer là số byte đã được trao đổi.
- fWait là tham số báo cho hàm có đợi cho thao tác vào ra hoàn tất hay không. Thí dụ, nếu fWait là TRUE thì hàm sẽ đợi đến khi thao tác vào ra hoàn tất mới quay trở lại.
- lpdwFlags là cờ kết quả của thao tác vào ra.

Nếu WSAGetOverlappedResult thành công, giá trị trả về là TRUE, nghĩa là thao tác vào ra hoàn tất, và lpcbTransfer là số byte đã trao đổi. Nếu hàm trả về FALSE có nghĩa là một trong các điều kiện sau đã xảy ra:

- Thao tác vào ra chưa hoàn tất.
- Thao tác vào ra hoàn tất nhưng có lỗi.
- Không thể xác định được trạng thái của thao tác vào ra do tham số đầu vào sai.

Đoạn mã sau minh họa việc sử dụng mô hình vào ra overlapped với phương thức xử lý hoàn thành thông qua sự kiện để xử lý việc nhận dữ liệu trên server:

```
void main(void)
{
```

```
  WSABUF DataBuf;
  char buffer[DATA_BUFSIZE];
  DWORD EventTotal = 0,
      RecvBytes=0,
      Flags=0;
  WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
  WSAOVERLAPPED AcceptOverlapped;
  SOCKET ListenSocket, AcceptSocket;

  // Bước 1:
  //  Khởi tạo WinSock và tạo ListenSocket
  ...

  // Bước 2:
  //  Chấp nhận kết nối
  AcceptSocket = accept(ListenSocket, NULL, NULL);

  // Bước 3:
  //  Thiết lập cấu trúc AccepOverlapped

  EventArray[EventTotal] = WSACreateEvent(); // Tạo đối tượng sự kiện
  ZeroMemory(&AcceptOverlapped, sizeof(WSAOVERLAPPED));
  AcceptOverlapped.hEvent = EventArray[EventTotal];
  // Thiết lập cấu trúc DataBuf
  DataBuf.len = DATA_BUFSIZE;
  DataBuf.buf = buffer;

  EventTotal++;

  // Bước 4:
  // Nhận dữ liệu
  if (WSARecv(AcceptSocket, &DataBuf, 1, &RecvBytes,
     &Flags, &AcceptOverlapped, NULL) == SOCKET_ERROR)
  {
   // Kiểm tra lỗi
   if (WSAGetLastError() != WSA_IO_PENDING)
   {
      // Error occurred
   }
}
  // Xử lý sự kiện trên cấu trúc overlapped
  while(TRUE)
  {
```

```
   DWORD    Index;
    // Bước 5:
    //  Đợi thao tác vào ra hoàn tất
    Index = WSAWaitForMultipleEvents(EventTotal,
       EventArray, FALSE, WSA_INFINITE, FALSE);
     // Bước 6:
    //  Thiết lập lại đối tượng sự kiện về trạng thái chưa báo hiệu
    WSAResetEvent(
       EventArray[Index - WSA_WAIT_EVENT_0]);
    // Bước 7:
    //  Xác định trạng thái của thao tác vào ra
    WSAGetOverlappedResult(AcceptSocket,
       &AcceptOverlapped, &BytesTransferred,
       FALSE, &Flags);
    // Kiểm tra kết nối đã bị đóng chưa, nếu không có byte nào nhận được nghĩa là đã
    // đóng
    if (BytesTransferred == 0)
    {
       printf("Closing socket %d\n", AcceptSocket);
       closesocket(AcceptSocket);
       WSACloseEvent(
          EventArray[Index - WSA_WAIT_EVENT_0]);
       return;
    }
    // Xử lý dữ liệu nhận được trong biến DataBuf
    ...
    // Bước 8:
    //  Nhận thêm dữ liệu từ socket
    Flags = 0;
    ZeroMemory(&AcceptOverlapped,
       sizeof(WSAOVERLAPPED));
    AcceptOverlapped.hEvent = EventArray[Index -
       WSA_WAIT_EVENT_0];
    DataBuf.len = DATA_BUFSIZE;
    DataBuf.buf = buffer;
    if (WSARecv(AcceptSocket, &DataBuf, 1, &RecvBytes, &Flags,
&AcceptOverlapped,
       NULL) == SOCKET_ERROR)
    {
       if (WSAGetLastError() != WSA_IO_PENDING)
       {
          // Lỗi
       }
```

```
    }
  }
}
```

Xử lý hoàn thành thông qua completion routine. Completion rotine là đoạn chương trình sẽ được hệ thống gọi khi thao tác vào ra hoàn tất. Completion routine sẽ được chạy trong luồng của lời gọi thao tác vào ra. Để xử lý kiểu này, ứng dụng phải truyền tham số là completion routine trong các lời gọi hàm vào ra. Nguyên mẫu của completion routine có dạng sau:

```
void CALLBACK CompletionRoutine(
  DWORD dwError,
  DWORD cbTransferred,
  LPWSAOVERLAPPED lpOverlapped,
  DWORD dwFlags
);
```

Khi thao tác vào ra hoàn tất, hệ thống sẽ triệu gọi CompletionRoutine và truyền các tham số sau

- dwError mô tả lỗi của thao tác vào ra.
- cbTransfered là số byte đã trao đổi.
- lpOverlapped là cấu trúc overlapped đã sử dụng cho thao tác vào ra
- dwFlags chứa cờ của thao tác.

Để hệ thống có thể gọi được CompletionRoutine, ứng dụng cần chuyển luồng chứa lời gọi sang trạng thái ***alertable wait state***. Hàm WaitForMultipleEvents có thể chuyển luồng hiện tại sang trạng thái đó, tuy nhiên hàm này yêu cầu đầu vào là ít nhất một đối tượng sự kiện. Khi sử dụng phương thức **completion routine** nghĩa là chương trình sẽ không sự dụng đối tượng sự kiện nào. Hàm SleepEx cũng có thể chuyển luồng hiện tại sang trạng thái ***alertable wait state*** và không cần đối tượng sự kiện nào.

```
DWORD SleepEx(
  DWORD dwMilliseconds,
  BOOL bAlertable
);
```

Đoạn mã sau đây minh họa việc sử dụng Completion Routine để xử lý vào ra trên một server đơn giản

```
SOCKET AcceptSocket, ListenSocket;
WSABUF DataBuf;
DWORD Flags, RecvBytes,Ret;
char buffer[DATA_BUFSIZE];

void main(void)
```

```
{
  WSAOVERLAPPED Overlapped;

  // Bước 1:
  // Khởi tạo WinSock, thiết lập ListenSocket
  ...

  // Bước 2:
  // Chấp nhận kết nối mới
  AcceptSocket = accept(ListenSocket, NULL, NULL);

  // Bước 3:
  // Khởi tạo cấu trúc overlapped

  Flags = 0;

  ZeroMemory(&Overlapped, sizeof(WSAOVERLAPPED));

  DataBuf.len = DATA_BUFSIZE;
  DataBuf.buf = buffer;
   // Step 4:
  //  Gửi yêu cầu vào ra với địa chỉ của Completion Routine

  if (WSARecv(AcceptSocket, &DataBuf, 1, &RecvBytes, &Flags, &Overlapped,
WorkerRoutine) == SOCKET_ERROR)
  {
     // Xử lý lỗi
     if (WSAGetLastError() != WSA_IO_PENDING)
     {
        printf("WSARecv() failed with error %d\n",
           WSAGetLastError());
        return;
     }
  }

  while(TRUE)
  {
     // Bước 5: Chuyển luồng hiện tại sang trạng thái alertable wait state
     Ret = SleepEx(INFINITE, TRUE);
     // Bước 6: Xử lý công việc còn lại sau khi hoàn tất thao tác vào ra
     if (Ret == WAIT_IO_COMPLETION)
     {
        continue;
```

```
    }
    else
    {

        return;
    }
  }
}
// Hàm call back được gọi khi thao tác vào ra hoàn tất
void CALLBACK WorkerRoutine(DWORD Error,
                DWORD BytesTransferred,
                LPWSAOVERLAPPED Overlapped,
                DWORD InFlags)
{
  DWORD SendBytes, RecvBytes;
  DWORD Flags;

  if (Error != 0 || BytesTransferred == 0)
  {
    // Đã có lỗi xảy ra hoặc socket bị đóng
    closesocket(AcceptSocket);
    return;
  }

  // Tại thời điểm này thao tác vào ra đã hoàn tất, ứng dụng có thể xử lý dữ liệu nhận
  // được trong biến DataBuf
  Flags = 0;

  ZeroMemory(&Overlapped, sizeof(WSAOVERLAPPED));

  DataBuf.len = DATA_BUFSIZE;
  DataBuf.buf = buffer;
   // Gửi trả dữ liệu nếu cần
   //  ...

}
```

Mô hình vào ra overlapped có ưu điểm là hiệu năng cao. Ứng dụng gửi bộ đệm chứa dữ liệu cần gửi hay nhận cho WinSock, và hệ thống sữ dụng bộ đệm này một cách trực tiếp thay vì phải sao chép nhiều lần như các mô hình vào ra khác. Hạn chế của mô hình này là nếu thực hiện theo phương thức xử lý sự kiện hoàn thành, thì mỗi luồng sẽ quản lý tối đa được 64 socket, nếu xử lý theo

completion routine, ứng dụng cần đặt luồng hiện tại sang trạng thái alertable wait state, nghĩa là không thể thực hiện được các tính toán khác.

Việc lập trình ứng dụng sử dụng các thư viện chuẩn của WinSock đôi khi phức tạp và khó tiếp cận hơn các thư viện hướng đối tượng. Các chương sau sẽ trình bày phương pháp lập trình mạng sử dụng các thư viện hướng đối tượng C++.

Chương 4. MFC Socket

Bài giảng số 7

- **Thời lượng: 3 tiết**
- **Tóm tắt nội dung:**
  - **Giới thiệu về MFC Socket**
  - **Sử dụng lớp CSocket**
    - **Khởi tạo CSocket**
    - **Kết nối đến máy khác**
    - **Chấp nhận kết nối từ máy khác**
    - **Gửi nhận dữ liệu**
    - **Đóng kết nối**
  - **Sử dụng lớp CAsyncSocket**
    - **Khởi tạo đối tượng**
    - **Xử lý các sự kiện**

4.1 Giới thiệu

Bộ thư viện MFC Socket hỗ trợ hai mô hình lập trình mạng được và đóng gói vào hai lớp:

- **CAsyncSocket**
  Lớp này đóng gói lại thư viện WinSock. CAsyncSocket dành cho các lập trình viên đã có kinh nghiệm lập trình mạng, và muốn tận dụng tính mềm dẻo của WinSock cùng với sự tiện lợi mà ngôn ngữ hướng đối tượng C++ mang lại. CAsyncSocket cũng đóng gói các sự kiện của WinSock và chuyển đến ứng dụng thông qua cơ chế thông điệp của Windows. CAsyncSocket hoạt động ở chế độ bất đồng bộ.
- **CSocket**
  Lớp này kế thừa từ CAsyncSocket, cung cấp một giao diện ở mức cao hơn nữa. CSocket dễ sử dụng và kế thừa nhiều phương thức từ CAsyncSocket. CSocket hoạt động ở chế độ đồng bộ.

Để sử dụng hai thư viện này, ứng dụng cần được phát triển trong môi trường Visual Studio C++.

4.2 CSocket

Sử dụng lớp CSocket tương đối đơn giản, các thao tác cơ bản gồm có: khởi tạo socket, kết nối đến socket khác, chấp nhận kết nối từ socket khác, gửi dữ liệu, nhận dữ liệu, đóng kết nối. Lớp CSocket đóng gói hoạt động của socket đồng bộ, do vậy mọi thao tác sẽ chặn luồng hiện tại cho đến khi hoàn tất.

4.2.1 Khởi tạo CSocket

Phương thức Create được dùng để khởi tạo đối tượng. Nguyên mẫu như sau

```
BOOL Create(
  UINT nSocketPort = 0,
  int nSocketType = SOCK_STREAM,
  LPCTSTR lpszSocketAddress = NULL
);
```

Trong đó:

- ***nSocketPort*** là cổng được chọn, mặc định nếu là 0 thì hệ điều hành sẽ chọn một cổng còn trống gán cho socket.
- ***nSocketType*** có hai giá trị là SOCK_STREAM tương ứng với giao thức TCP và SOCK_DATAGRAM tương ứng với giao thức UDP.
- ***lpszSocketAddress*** là địa chỉ của giao diện mạng mà socket sẽ gắn vào. Trong trường hợp máy tính có nhiều giao diện mạng thì tham số này sẽ cho phép lựa chọn giao diện cụ thể hoặc tất cả các giao diện.

Thí dụ sau đây sẽ khởi tạo đối tượng CSocket s với các tham số mặc định:

```
CSocket     s;
s. Create();     // Tạo socket với cổng mặc định
                 // hoặc
s. Create(80);// Tạo socket ở cổng 80
```

4.2.2 Kết nối đến máy khác

Phương thức Connect sẽ được dùng để nối đến socket khác. Có hai phương thức chồng.

```
BOOL Connect(
  LPCTSTR lpszHostAddress,
  UINT nHostPort
);
BOOL Connect(
  const SOCKADDR* lpSockAddr,
  int nSockAddrLen
);
```

Phương thức đầu tiên đơn giản hơn, nhận hai tham số, tham số thứ nhất là địa chỉ máy đích, tham số thứ hai là cổng cần kết nối. CSocket làm toàn bộ việc phân giải tên miền và lựa chọn địa chỉ hộ người lập trình. Thí dụ:

```
CSocket       s;
s.Create;
s.Connect(www.google.com.vn, 80);
```

4.2.3 Chấp nhận kết nối từ máy khác

Nếu ứng dụng là server, hàm Accept sẽ được sử dụng để chấp nhận kết nối từ máy khác.

```
virtual BOOL Accept(
  CSocket& rConnectedSocket,
  SOCKADDR* lpSockAddr = NULL,
  int* lpSockAddrLen = NULL
);
```

Thí dụ:

```
CSocket        connectedSocket;
listeningSocket.Accept(connectedSocket);
// Gửi nhận dữ liệu trên connectedSocket
// …
```

4.2.4 Gửi dữ liệu

Ứng dụng có thể sử dụng phương thức Send để gửi dữ liệu . Nguyên mẫu của phương thức như sau:

```
virtual int Send(
  const void* lpBuf,
  int nBufLen,
  int nFlags = 0
);
```

Trong đó:

- ***lpBuf*** là địa chỉ bộ đệm chứa dữ liệu cần gửi.
- ***nBufLen*** là số byte cần gửi
- ***nFlags*** là cờ gửi. Trên Windows giá trị duy nhất có thể có của nFlags là MSG_OOB tương ứng với việc gửi dữ liệu OOB (Out Of Band).

Giá trị trả về của hàm là số byte gửi được. Hàm Send là một hàm đồng bộ, và sẽ không trở về cho đến khi việc gửi hoàn tất hoặc có lỗi.

Thí dụ:

```
…
char buff[]=”Hello Network Programming”;
connectedSocket. Send(buff,strlen(buff));
…
```

4.2.5 Nhận dữ liệu

Ứng dụng sẽ sử dụng phương thức Receive để nhận dữ liệu từ socket.

```
virtual int Receive(
  void* lpBuf,
  int nBufLen,
```

```
  int nFlags = 0
);
```

Trong đó:

- ***lpBuf*** là địa chỉ bộ đệm chứa dữ liệu sẽ nhận được.
- ***nBufLen*** là kích thước bộ đệm theo byte.
- ***nFlags*** là cờ nhận, có thể nhận một hoặc cả hai giá trị sau
  - MSG_PEEK: Dữ liệu ứng dụng nhận về sẽ không bị xóa khỏi bộ đệm hệ thống.
  - MSG_OOB: Nhận dữ liệu OOB

Giá trị trả về là số byte nhận được.

Thí dụ:

```
…
char buff[1024];
int buflen = 1024, nBytesReceived;
nBytesReceived = connectedSocket. Receive(buff,1024);
…
```

4.2.6 Đóng kết nối

Ứng dụng sẽ đóng kết nối đang có bằng phương thức **Close().** Thí dụ

```
connectedSocket.Close()
```

4.2.7 Xây dựng Client bằng CSocket

Đoạn chương trình sau sẽ sử dụng lớp CSocket để nối đến máy chủ ở địa chỉ www.google.com và gửi một truy vấn HTTP.

```
…
CSocket sk;
unsigned char buff[1024];
char * request = "GET / HTTP/1.0\r\nHost:www.google.com\r\n\r\n";
int len = 0;
sk.Create();
sk.Connect(www.google.com,80);
sk.Send(request,strlen(request));
len = sk.Receive(buff,1024);
buff[len] = 0;
printf("%s",buff);
…
```

4.2.8 Xây dựng Server bằng CSocket

Đoạn chương trình sau sẽ sử dụng CSocket để xây dựng một Server đơn giản

```
…
CSocket         listen,connect;
char            * buff = “Hello Network Programming”;
listen.Create(80);
listen.Listen();
listen.Accept(connect);
connect.Send(buff,strlen(buff));
connect.Close();
…
```

4.3 CAsyncSocket

CAsyncSocket là lớp đóng gói hoạt động của socket bất đồng bộ. Các hàm vào ra trở lại ngay lập tức và ứng dụng sẽ phải xử lý kết quả vào ra sau đó. Các hàm vào ra có cú pháp tương tự như CSocket và sẽ không đề cập đến ở đây nữa. Lớp này cũng cung cấp khá nhiều phương thức ảo. Mỗi phương thức ảo sẽ tương ứng với một sự kiện của WinSock. Để có thể sử dụng đối tượng CAsyncSocket, ứng dụng cần kế thừa từ lớp CAsyncSocket và xây dựng các phương thức chồng lên các phương thức ảo xử lý sự kiện với socket. Các phương thức thường được chồng là:

- **OnAccept**

  Phương thức này sẽ được gọi mỗi khi có yêu cầu kết nối.

- **OnClose**
  Phương thức này sẽ được gọi mỗi khi socket đầu kia bị đóng.
- **OnSend**

  Phương thức này được gọi khi socket có thể gửi dữ liệu.

- **OnReceive**
  Phương thức này được gọi khi socket nhận được dữ liệu và chờ ứng dụng xử lý
- **OnConnect**
  Phương thức này được gọi khi yêu cầu kết nối được chấp nhận và socket đã sẵn sàng để gửi nhận dữ liệu.

4.3.1 Khởi tạo đối tượng CAsyncSocket

Phương thức sau sẽ khởi tạo một đối tượng CAsyncSocket

```
BOOL Create(
  UINT nSocketPort = 0,
  int nSocketType = SOCK_STREAM,
  long lEvent = FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT | FD_CLOSE,
  LPCTSTR lpszSocketAddress = NULL
```

```
);
```

So với CSocket, hàm khởi tạo có thêm tham số *lEvent* chính là mặt nạ chứa các sự kiện mà đối tượng muốn nhận từ hệ thống.

4.3.2 Xử lý các sự kiện

CAsyncSocket có các sự kiện tương ứng với các phương thức ảo của lớp. Để xử lý sự kiện, ứng dụng cần chồng phương thức lên các phương thức ảo đó. Thí dụ để bắt sự kiện kết nối hoàn tất (bất kể thành công hay thất bại). Ứng dụng cần xây dựng lớp kế thừa lớp CAsynSocket và chồng phương thức OnConnect. Nếu ứng dụng muốn gửi dữ liệu đi, nó phải chồng phương thức OnSend, phương thức này sẽ được gọi mỗi khi kết nối trên socket đó có thể gửi dữ liệu. Sau đó ứng dụng sẽ gọi phương thức Send() để thực sự gửi dữ liệu. Tương tự như vậy, muốn nhận dữ liệu, ứng dụng sẽ phải đợi cho đến khi OnReceive được gọi. Ứng dụng sẽ gọi Receive để thực sự nhận dữ liệu về.

Thí dụ sau đây sẽ minh họa việc xây dựng một Client dùng CAsyncSocket.

```
// Khai báo lớp MySocket
class MySocket : public CAsyncSocket
{
public:
        char buff[128];
        unsigned int buff_len;
        bool        bSendable;
        MySocket();
        virtual ~MySocket();
        virtual void OnAccept(int nErrorCode);
        virtual void OnClose(int nErrorCode);
        virtual void OnConnect(int nErrorCode);
        virtual void OnReceive(int nErrorCode);
        virtual void OnSend(int nErrorCode);
};

// MySocket
MySocket::MySocket()
{
        buff_len = 0; // Chưa có gì để gửi
        memset(buff,0,128);
        bSendable = 0;
}
MySocket::~MySocket()
{
}
```

```
// Cài đặt các phương thức của MySocket

void MySocket::OnAccept(int nErrorCode)
{
        CAsyncSocket::OnAccept(nErrorCode);
}
void MySocket::OnClose(int nErrorCode)
{
        // Xử lý sự kiện khi kết nối bị đóng
        CAsyncSocket::OnClose(nErrorCode);
}

void MySocket::OnConnect(int nErrorCode)
{
        // Xử lý sự kiện khi việc thực hiện kết nối hoàn tất
        if (nErrorCode==0)
            bSendable = 1;
        CAsyncSocket::OnConnect(nErrorCode);
}

void MySocket::OnReceive(int nErrorCode)
{
        // Nhận dữ liệu và hiển thị
        buf_len = Receive(buff,128);
        buff[buf_len] = 0;
        printf("%s",buff);
        buf_len = 0;
        CAsyncSocket::OnReceive(nErrorCode);
}

void MySocket::OnSend(int nErrorCode)
{
        // Gửi dữ liệu
        Send(buff,buf_len);
        buf_len = 0;
        CAsyncSocket::OnSend(nErrorCode);
}
...
// Đoạn chương trình chính
char            str[] = "Hello World";
MySocket        socket;
socket.Create();
socket.Connect("www.google.com",80);
```

```
// Đợi sự kiện kết nối hoàn tất trong OnConnect
...
// Gửi dữ liệu
if (bSendable = 1)
        sk.Send(str,strlen(str));
```

Chương 5. NET Socket

Bài giảng số 8

- **Thời lượng: 3 tiết**
- **Tóm tắt nội dung:**
  - **Giới thiệu các lớp quan trọng trong NameSpace System.Net và System.Net.Socket**
  - **Xây dựng chương trình phía máy chủ sử dụng TCP.**
  - **Xây dựng chương trình phía máy khách sử dụng TCP.**
  - **Xây dựng chương trình phía máy chủ sử dụng UDP.**
  - **Xây dựng chương trình phía máy khác sử dụng UDP.**

5.1. Giới thiệu về NameSpace System.Net và System.Net.Sockets

NameSpace là môt tập hợp các lớp có liên hệ gần gũi với nhau trong bộ thư viện .NET. Hai NameSpace quan trọng hỗ trợ việc lập trình mạng trong .NET là System.Net và System.Net.Sockets. Mỗi namespace trên cung cấp khá nhiều lớp hỗ trợ, trong đó các lớp thông dụng là IPAddress, IPEndPoint, DNS…

- Lớp IPAdress: lớp quản lý các thao tác liên quan đến địa chỉ IP. Lớp này có các trường đặc biệt sau:
  - Any: Đây là địa chỉ chỉ ra rằng Server phải lắng nghe trên tất cả các Card mạng.
  - Broadcast: Địa chỉ quảng bá của mạng hiện tại.
  - Loopback: Địa chỉ lặp.
  - AddressFamily: họ địa chỉ IP hiện tại.

  Một số phương thức cần chú ý liên quan đến lớp này:

  - Hàm khởi tạo:
    - IPAddress(Byte[])
    - IPAddress(Int64)
  - IsLoopback: Cho biết địa chỉ hiện tại có phải địa chỉ lặp không.
  - Parse: Chuyển IP dạng xâu về IP chuẩn.
  - ToString: Trả về địa chỉ IP dưới dạng dạng xâu ký tự.
  - TryParse: Kiểm tra IP ở dạng xâu có hợp lệ không.
- Lớp IPEndPoint: lớp này đóng gói thông tin cần thiết về địa chỉ và cổng của dịch vụ cần kết nối đến.Một số phương thức cần chú ý:
  - Phương thức khởi tạo:
    - IPEndPoint (Int64, Int32)
    - IPEndPoint (IPAddress, Int32)
  - Create: Tạo một EndPoint từ một địa chỉ SocketAddress.
  - ToString : Trả về địa chỉ IP và số hiệu cổng theo khuôn dạng "địa chỉ: cổng", ví dụ: 192.168.1.1:8080

- Lớp DNS: lớp này hỗ trợ các thao tác phân giải tên miền. Một số thành phần của lớp:
    - HostName: Cho biết tên của máy được phân giải.
    - GetHostAddress: Trả về tất cả IP của một tên miền tương ứng.
    - GetHostEntry: Thực hiện phân giải tên hoặc địa chỉ truyền vào và trả về đối tượng kiểu IPHostEntry.

- GetHostName: Lấy về tên của máy tính cục bộ.

NameSpace System.Net.Sockets cũng cung cấp các lớp thông dụng: TcpClient, UdpClient, TcpListener, Socket, NetworkStream, …

Phương thức sau sẽ được dùng để khởi tạo một socket trên .NET

```
Socket(AddressFamily af, SocketType st, ProtocolType pt)
```

Trong đó các tham số thường được sử dụng kèm với nhau như sau:

SocketType	**Protocoltype**	**Description**
Dgram	Udp	Connectionless communication
Stream	Tcp	Connection-oriented
Raw	Icmp	Internet Control Message
Raw	Raw	Plain IP packet communication

Đoạn chương trình sau sẽ minh họa việc khởi tạo các lớp thông dụng.

```
using System.Net;
using System.Net.Sockets;
class SockProp    {
public static void Main()
{
        IPAddress ia = IPAddress.Parse("127.0.0.1");
        IPEndPoint ie = new IPEndPoint(ia, 8000);
        Socket test = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
        ProtocolType.Tcp);
        Console.WriteLine("AddressFamily: {0}", test.AddressFamily);
        Console.WriteLine("SocketType: {0}", test.SocketType);
        Console.WriteLine("ProtocolType: {0}",  test.ProtocolType);
        Console.WriteLine("Blocking: {0}", test.Blocking);
        test.Blocking = false;
        Console.WriteLine("new Blocking: {0}", test.Blocking);
        Console.WriteLine("Connected: {0}", test.Connected); test.Bind(ie);
         IPEndPoint iep = (IPEndPoint)test.LocalEndPoint;
        Console.WriteLine("Local EndPoint: {0}", iep.ToString());
        test.Close(); Console.ReadKey();
}
```

5.2. Chương trình cho phía máy chủ sử dụng giao thức TCP

Các công việc cần thiết khi viết chương trình cho máy server:

- Tạo một Socket

- Liên kết với một IPEndPoint cục bộ
- Lắng nghe kết nối
- Chấp nhận kết nối
- Gửi nhận dữ liệu theo giao thức ñã thiết kế
- Đóng kết nối sau khi đã hoàn thành và trở lại trạng thái lắng nghe chờ kết nối mới.

Đoạn chương trình minh họa:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net;
using System.Net.Sockets;
class Server{
static void Main(string[] args) {
IPEndPoint iep = new IPEndPoint(IPAddress.Any, 8888);
Socket server = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
server.Bind(iep);
server.Listen(10);
Console.WriteLine("Cho ket noi tu client");
Socket client = server.Accept();
Console.WriteLine("Chap nhan ket noi tu:{0}",client.RemoteEndPoint.ToString());
string s = "Chao ban den voi Server";
//Chuyen chuoi s thanh mang byte
byte[] data = new byte[1024];
data = Encoding.ASCII.GetBytes(s);
//Gửi dữ liệu
 client.Send(data,data.Length,SocketFlags.None);
while (true) {
data = new byte[1024];
int recv = client.Receive(data);
if (recv == 0) break;
//Chuyen mang byte Data thanh chuoi va in ra man hinh s = Encoding.ASCII.GetString(data, 0, recv); Console.WriteLine("Clien gui len:{0}", s);
//Neu chuoi nhan duoc la Quit thi thoat if (s.ToUpper().Equals("QUIT")) break;
//Gui tra lai cho client chuoi s s = s.ToUpper();
data = new byte[1024];
data = Encoding.ASCII.GetBytes(s);
client.Send(data, data.Length, SocketFlags.None);
}
client.Shutdown(SocketShutdown.Both);
client.Close();
```

```
server.Close();
}
}
```

5.3. Chương trình cho phía máy khách sử dụng giao thức TCP

Các công việc cần thực hiện tại chương trình phía máy khách

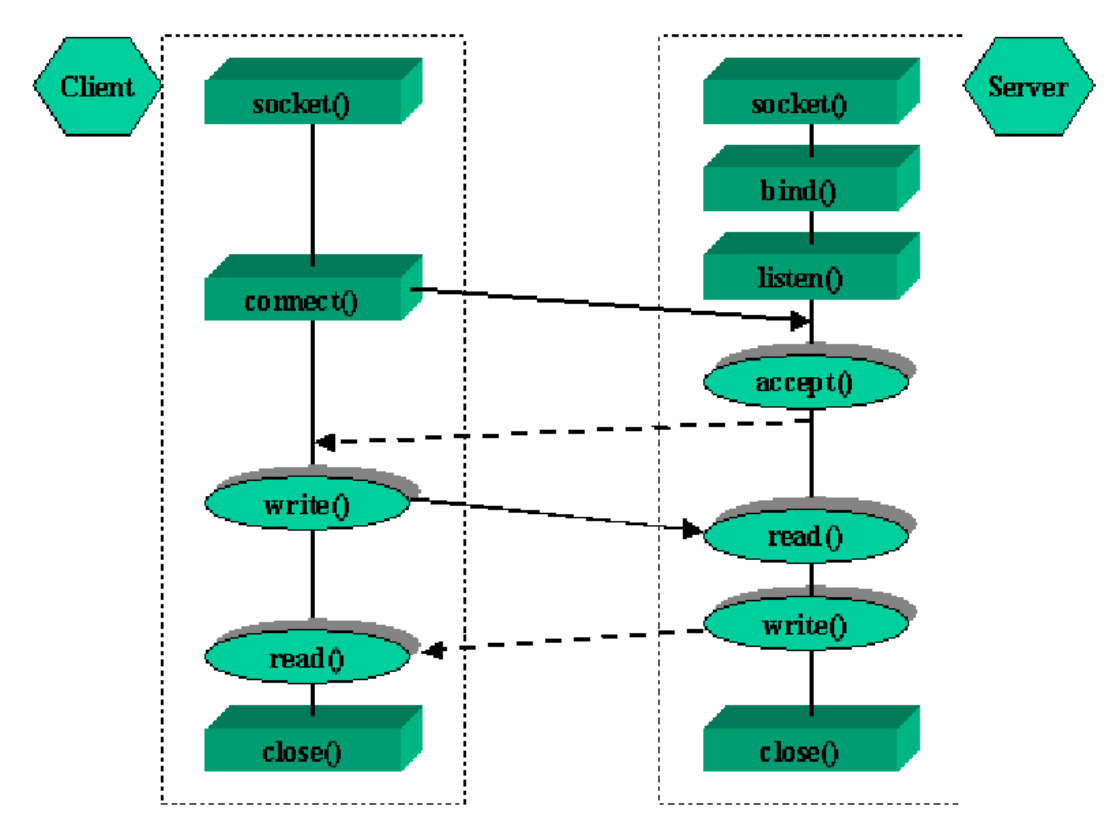
- Xác định địa chỉ của Server
- Tạo Socket
- Kết nối đến Server
- Gửi nhận dữ liệu theo giao thức đã thiết kế
- Đóng Socket

Đoạn chương trình minh họa

```
IPEndPoint ipep = new IPEndPoint(Ipaddress.Parse("127.0.0.1"), 8888);
Socket server = new Socket(AddressFamily.InterNetwork,SocketType.Stream,
ProtocolType.Tcp);
server.Connect(ipep);
Chương trình Client:
using System;
using System.Collections.Generic;
using System.Text;
using System.Net;
using System.Net.Sockets;

class Client {
static void Main(string[] args) {
IPEndPoint iep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 2008);
Socket client = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
client.Connect(iep);
byte[] data = new byte[1024];
int recv = client.Receive(data);
string s = Encoding.ASCII.GetString(data, 0, recv); Console.WriteLine("Server gui:{0}",
s);
string input;
while (true) {
input = Console.ReadLine();
//Chuyen input thanh mang byte gui len cho server data = new byte[1024];
data = Encoding.ASCII.GetBytes(input);
client.Send(data, data.Length, SocketFlags.None);
if (input.ToUpper().Equals("QUIT")) break;
data = new byte[1024];
recv = client.Receive(data);
```

```
s = Encoding.ASCII.GetString(data, 0, recv); Console.WriteLine("Server gui:{0}", s);
}
client.Disconnect(true);
client.Close();
}
}
```



Hình 8. Trình tự gửi nhận dữ liệu theo giao thức TCP

5.4 Chương trình phía máy chủ sử dụng UDP

Các công việc cần thực hiện:

- Tạo một Socket
- Liên kết với một IPEndPoint cục bộ
- Gửi nhận dữ liệu theo giao thức ñã thiết kế
- Đóng Socket

Đoạn chương trình minh họa

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net;
using System.Net.Sockets;
class Program {
static void Main(string[] args) {
```

```
IPEndPoint     iep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8888);
Socket server = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
ProtocolType.Udp);
server.Bind(iep);
//tao ra mot Endpot tu xa de nhan du lieu ve
IPEndPoint     RemoteEp = new IPEndPoint(IPAddress.Any, 0);
EndPoint       remote=(EndPoint)RemoteEp;
byte[] data = new byte[1024];
int recv = server.ReceiveFrom(data, ref remote); string s =
Encoding.ASCII.GetString(data, 0, recv); Console.WriteLine("nhan ve tu Client:{0}", s);
data = Encoding.ASCII.GetBytes("Chao client");
server.SendTo(data, remote);
while (true) {
data=new byte[1024];
recv = server.ReceiveFrom(data, ref remote); s = Encoding.ASCII.GetString(data, 0,
recv); if (s.ToUpper().Equals("QUIT")) break; Console.WriteLine(s);
data=new byte[1024];
data=Encoding.ASCII.GetBytes(s);
server.SendTo(data,0,data.Length,SocketFlags.None,remote);
}
server.Close();
}
```

5.5 Chương trình cho máy khách sử dụng UDP

Các công việc cần thực hiện

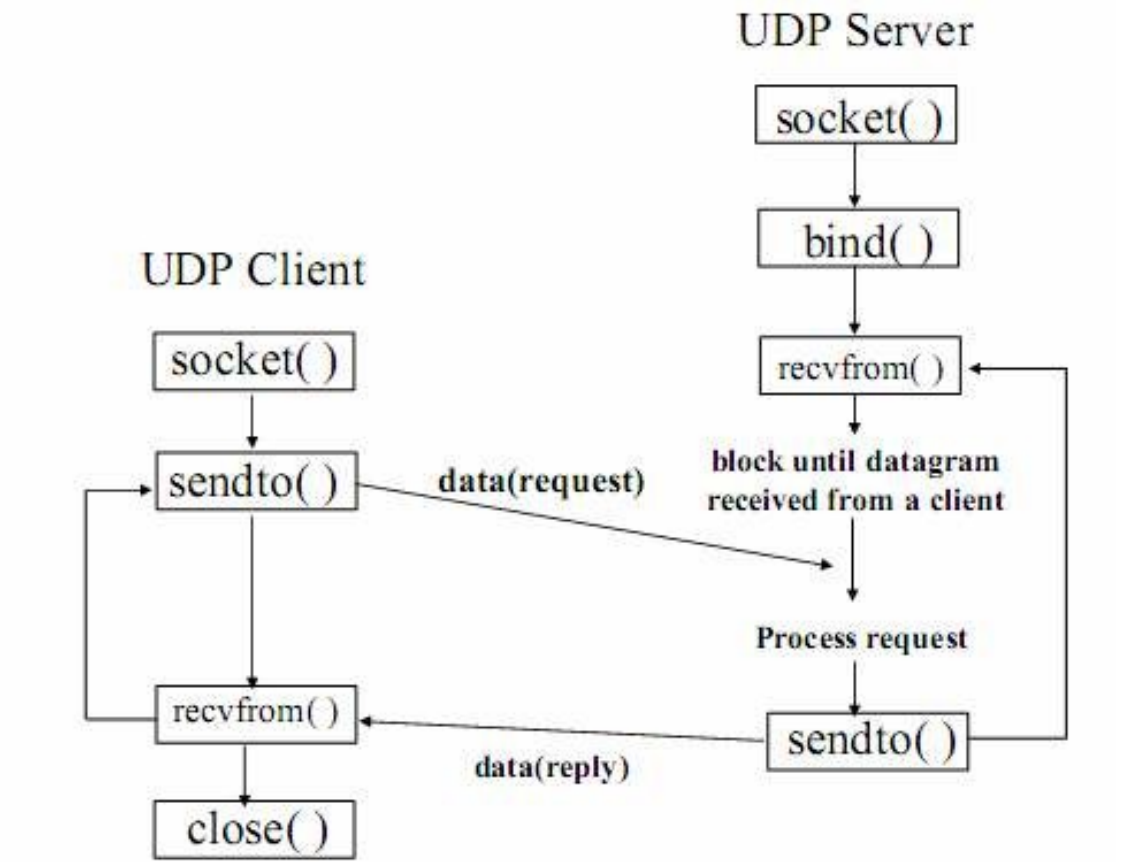
- Xác định địa chỉ Server
- Tạo Socket
- Gửi nhận dữ liệu theo giao thức ñã thiết kế
- Đóng Socket

Đoạn chương trình minh họa

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net;
using System.Net.Sockets;

class Program {
static void Main(string[] args) {
IPEndPoint iep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 2008);
Socket client = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
ProtocolType.Udp);
String s = "Chao server";
```

```
byte[] data = new byte[1024];
data = Encoding.ASCII.GetBytes(s);
client.SendTo(data, iep);
EndPoint remote = (EndPoint)iep;
data = new byte[1024];
int recv = client.ReceiveFrom(data, ref remote); s = Encoding.ASCII.GetString(data, 0,
recv); Console.WriteLine("Nhan ve tu Server{0}",s);
while (true) {
s = Console.ReadLine();
data=new byte[1024];
data = Encoding.ASCII.GetBytes(s);
client.SendTo(data, remote);
if (s.ToUpper().Equals("QUIT")) break;
data = new byte[1024];
recv = client.ReceiveFrom(data, ref remote);
s = Encoding.ASCII.GetString(data, 0, recv); Console.WriteLine(s);
}
client.Close();
}
```



Hình 9. Trình tự gửi nhận dữ liệu theo giao thức UDP