BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC TÂY BẮC

GIÁO TRÌNH

LẬP TRÌNH MẠNG

NGUYỄN DUY HIẾU
MAI VĂN TÁM

SƠN LA, NĂM 2019

LỜI NÓI ĐẦU

Trong những năm gần đây, lập trình mạng luôn là một trong những nội dung quan trọng trong lĩnh vực công nghệ phần mềm. Nhờ sự phát triển vượt bậc trong lĩnh vực mạng máy tính, những phần mềm máy tính dùng cho doanh nghiệp hiện nay sử dụng rất nhiều trên môi trường mạng đặc biệt là Internet. Mạng máy tính là nơi các kỹ thuật liên quan tới mô hình khách/chủ, mô hình phân tán hay mô hình hợp tác được triển khai. Các ứng dụng mạng xử lý tập trung hoặc phân tán, tận dụng tối đa sức mạnh của các hệ thống phần cứng để mang lại hiệu quả cao.

Giáo trình này phục vụ giảng dạy và học tập học phần *Lập trình mạng* tại Trường Đại học Tây Bắc. Trong nội dung, ngoài những kiến thức cơ bản về mạng máy tính có liên quan, chúng tôi sẽ hướng dẫn cách thức làm việc với kỹ thuật lập trình socket với giao thức TCP, UDP và kỹ thuật lập trình phân tán RMI. Trong từng phần kiến thức, giáo trình cũng cung cấp các ví dụ minh họa. Đây là những ví dụ hết sức cơ bản, giúp bạn đọc hiểu được cách thức mà các ứng dụng mạng hoạt động. Từ những ví dụ này, độc giả có thể tự phát triển để tạo ra các ứng dụng phức tạp hơn, mạnh mẽ hơn.

Chúng tôi lựa chọn ngôn ngữ lập trình Java để trình bày về các kỹ thuật liên quan tới lập trình mạng cũng như các ví dụ minh họa. Việc lựa chọn ngôn ngữ lập trình Java không chỉ bởi Java hiện nay luôn là một trong những ngôn ngữ hàng đầu để phát triển phần mềm mà còn bởi Java là ngôn ngữ vốn sinh ra để giải quyết các vấn đề liên quan tới các ứng dụng mạng.

Giáo trình không tránh khỏi những sơ suất. Chúng tôi mong nhận được các ý kiến đóng góp quý báu của quý thầy cô và các bạn sinh viên để hoàn thiện giáo trình hơn nữa.

Chúng tôi xin chân thành cảm ơn.

NHÓM TÁC GIẢ

MỤC LỤC

DANH MỤC HÌNH ẢNH

CHƯƠNG 1. CÁC KHÁI NIỆM CƠ BẢN VỀ MẠNG MÁY TÍNH

1.1 Mạng máy tính

Một mạng máy tính là một tập hợp các máy tính và các thiết bị. Các máy tính và các thiết bị trên mạng có thể gửi và nhận dữ liệu với các máy tính và các thiết bị khác. Với mạng được kết nối bằng dây dẫn, các bit dữ liệu sẽ được biến đổi thành các sóng điện từ di chuyển dọc theo dây dẫn. Các mạng không dây truyền dữ liệu bằng cách sử dụng sóng vô tuyến. Và với những đường truyền có khoảng cách lớn, dữ liệu được truyền đi bằng cách sử dụng cáp quang. Cáp quang sử dụng các sóng ánh sáng với độ dài bước sóng khác nhau để truyền dữ liệu.

Mỗi một thiết bị trên một mạng được gọi là một *nút mạng* (node). Hầu hết các nút là các máy tính, tuy nhiên các nút cũng có thể là các máy in, router, cầu nối, gateway (thiết bị nối ghép hai mạng cục bộ không cùng họ với nhau, hoặc mạng cục bộ với một mạng diện rộng, với một máy tính mini hay máy tính lớn…), các thiết bị cuối câm (thiết bị cuối không có bộ xử lý trung tâm và các ổ đĩa - dumb terminal). Ta sẽ dùng thuật ngữ *nút* để nói đến bất kỳ thiết bị nào trên mạng và dùng thuật ngữ *host* để nói đến một nút là một máy tính đa năng. Mỗi một nút mạng có một địa chỉ. Một địa chỉ là một chuỗi các byte xác định duy nhất một nút.

Địa chỉ được gán cho từng nút mạng sẽ khác nhau đối với các mạng khác nhau. Các địa chỉ Ethernet được gắn vào phần cứng vật lý Ethernet. Các nhà sản xuất phần cứng Ethernet sử dụng mã nhà sản xuất được chỉ định trước để đảm bảo không có xung đột giữa địa chỉ trong phần cứng của họ và địa chỉ của phần cứng của nhà sản xuất khác. Mỗi nhà sản xuất chịu trách nhiệm đảm bảo rằng không có bất kỳ hai card mạng Ethernet nào có cùng địa chỉ. Các địa chỉ Internet thường được gán cho một máy tính bởi Nhà cung cấp dịch vụ Internet (Internet Service Provider - ISP).

Các mạng máy tính hiện đại là các mạng chuyển mạch gói (packet-switched): dữ liệu truyền trên mạng được chia nhỏ thành các đoạn được gọi là các gói (packet). Mỗi một gói chứa thông tin về bên gửi và bên nhận. Một số ưu điểm của việc chia nhỏ dữ liệu thành các gói được gán địa chỉ là:

+ Các gói tin từ nhiều nguồn trao đổi thông tin khác nhau có thể được truyền đi trên cùng một đường dây và sẽ làm giảm giá thành cho việc xây dựng các mạng. Các máy tính có thể chia sẻ chung một đường truyền mà không sợ bị can nhiễu lẫn nhau.

+ Các mã kiểm tra tổng (checksum) có thể được sử dụng để phát hiện gói tin có bị hư hại trong quá trình truyền tin hay không.

Để các mạng máy tính hoạt động được ta cần phải có những quy tắc hoạt động, đó chính là *các giao thức* (protocol). Một giao thức là một tập hợp chính xác các luật định nghĩa cách thức các máy tính giao tiếp với nhau: khuôn dạng của các địa chỉ, cách chia dữ liệu được thành các gói… Ví dụ:

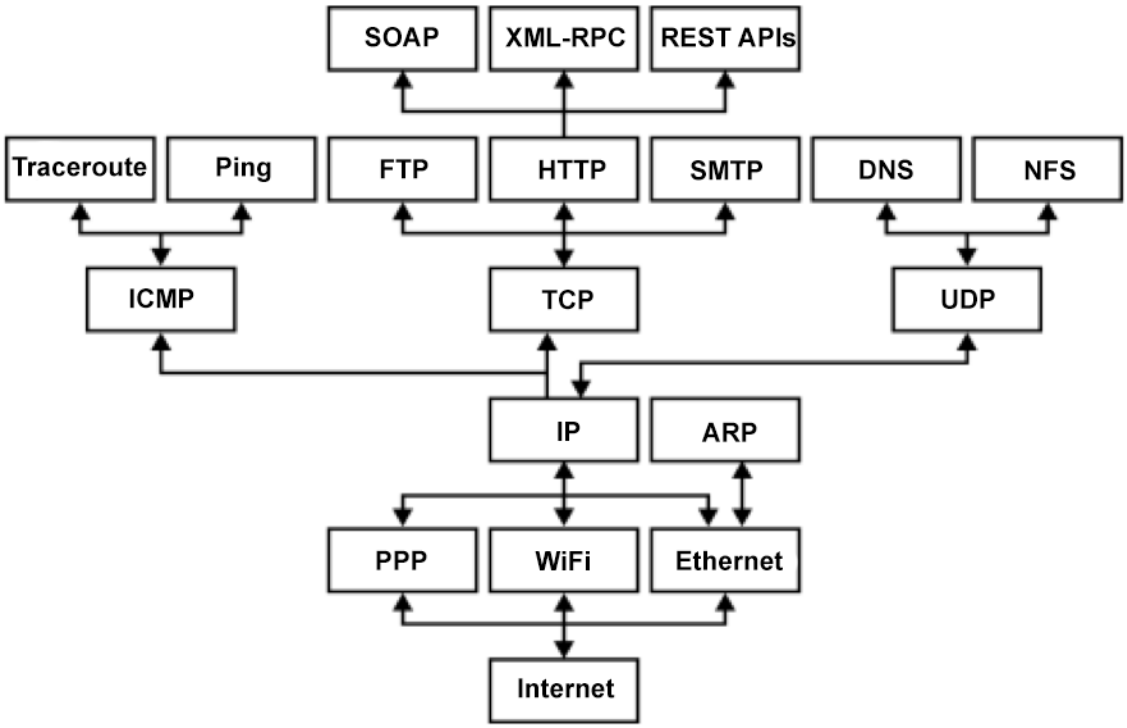
+ Giao thức HTTP (Hypertext Transfer Protocol) quy định cách thức trao đổi thông tin giữa các trình duyệt web (web browsers) với máy chủ dịch vụ web (webserver).

+ Chuẩn IEEE 802.3 quy định cách mã hóa các bit thành các tín hiệu điện trên từng loại dây dẫn cụ thể.

Các chuẩn giao thức mở, đã được công bố cho phép phần mềm và thiết bị từ các nhà cung cấp khác nhau giao tiếp với nhau. Ví dụ một máy chủ web không cần phải quan tâm người dùng sẽ sử dụng hệ điều hành nào trên thiết bị của họ, chẳng hạnh như Windows, Unix, Android, iOS… vì tất cả các thiết bị này đều sử dụng chung giao thức HTTP.

1.2. Các lớp của một mạng

Truyền dữ liệu trên một mạng là một quá trình phức tạp. Để người phát triển ứng dụng và để người sử dụng không cần nhìn thấy sự phức tạp của quá trình truyền dữ liệu, các thành phần khác nhau của một mạng truyền thông được chia thành nhiều lớp. Mỗi một lớp biểu diễn một mức khác nhau của việc trừu tượng hóa giữa phần cứng vật lý (dây dẫn, điện…) và thông tin được truyền.

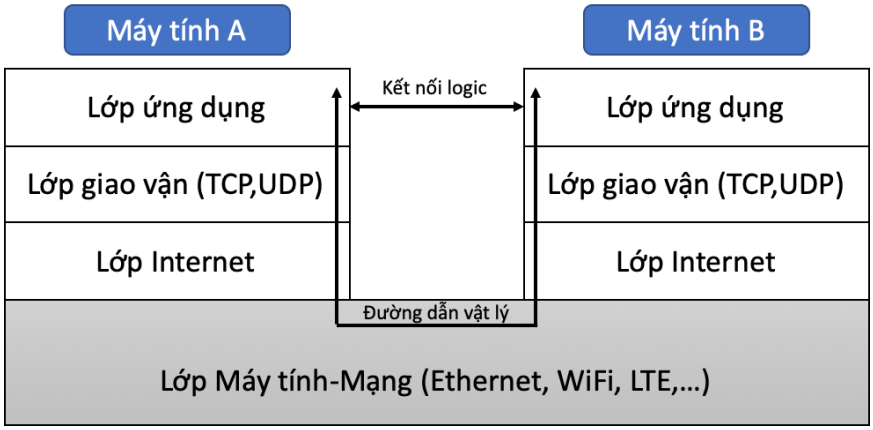


Hình 1.1: Các giao thức trong các lớp khác nhau của một mạng

Về mặt lý thuyết, mỗi một lớp chỉ trao đổi với các lớp kề ngay trên và kề ngay dưới. Việc tách một mạng thành các lớp cho phép ta sửa đổi thậm chí là thay thế phần mềm trong một lớp mà không ảnh hưởng đến các lớp khác miễn sao cho các giao diện giữa các lớp không thay đổi.

Sơ đồ trong *Hình 1.1* trình bày một mô hình phân tầng của các giao thức có thể có trong một mạng. Trong khi các giao thức các lớp ở giữa tương đối ổn định trong mạng Internet thì các giao thức trên đỉnh và dưới đáy thay đổi rất nhiều.

Tồn tại nhiều mô hình chia lớp khác nhau như mô hình OSI, mô hình TCP/IP. Mỗi mô hình phù hợp với các yêu cầu của một kiểu mạng cụ thể. Trong tài liệu này ta sẽ sử dụng mô hình TCP/IP, đây là mô hình chuẩn bốn lớp phù hợp với mạng Internet. Sơ đồ sau sẽ trình bày cấu trúc của mô hình TCP/IP.



Hình 1.2: Các lớp trong mô hình TCP/IP

Các lớp trong mô hình TCP/IP bao gồm:

+ Lớp ứng dụng (Application Layer).

+ Lớp giao vận (Transport Layer) sử dụng các giao thức TCP hoặc UDP.

+ Lớp Internet (Internet Layer) sử dụng giao thức IP.

+ Lớp máy tính-mạng (Host-To-Network Layer) sử dụng các giao thức Ethernet, WiFi, LTE…

Một ứng dụng được chạy trong lớp ứng dụng và chỉ trao đổi thông tin với lớp giao vận. Lớp giao vận chỉ trao đổi thông tin với lớp ứng dụng và lớp Internet. Lớp Internet, đến lượt mình, chỉ trao đổi thông tin với lớp giao vận và lớp máy tính-mạng và không trao đổi trực tiếp với lớp ứng dụng. Lớp máy tính-mạng chuyển dữ liệu thông qua các dây dẫn, cáp quang hoặc các đường truyền vật lý khác tới lớp máy tính-mạng trên hệ thống từ xa khác trong mạng. Lớp máy tính-mạng trên hệ thống từ xa sau đó sẽ chuyển dữ liệu lên các lớp bên trên tới lớp ứng dụng trong hệ thống này. Ta sẽ tìm hiểu chức năng của mỗi lớp ở các phần dưới đây.

1.2.1 Lớp máy tính-mạng

Lớp máy tính-mạng (lớp vật lý) quy định cách thức một giao diện mạng cụ thể. Chẳng hạn như một card Ethernet hoặc một ăng-ten WiFi gửi các IP datagram qua kết nối vật lý của nó tới mạng cục bộ và tới mạng diện rộng.

Lớp máy tính-mạng được xây dựng bởi phần cứng kết nối các máy tính với nhau (dây dẫn, cáp quang, sóng vô tuyến) và đôi khi còn được gọi là lớp vật lý của một mạng. Các lập trình viên sử dụng ngôn ngữ Java không cần phải quan tâm tới lớp này trừ khi có các vấn đề về kỹ thuật cần phải khắc phục. Các lập trình viên chỉ cần quan tâm đến hiệu suất của mạng, ví dụ khách hành của chúng ta sử dụng một mạng cáp quang tốc độ cao ta sẽ cần phải thiết kế một giao thức và các ứng dụng phù hợp với tốc đô của mạng bên phía khách hàng.

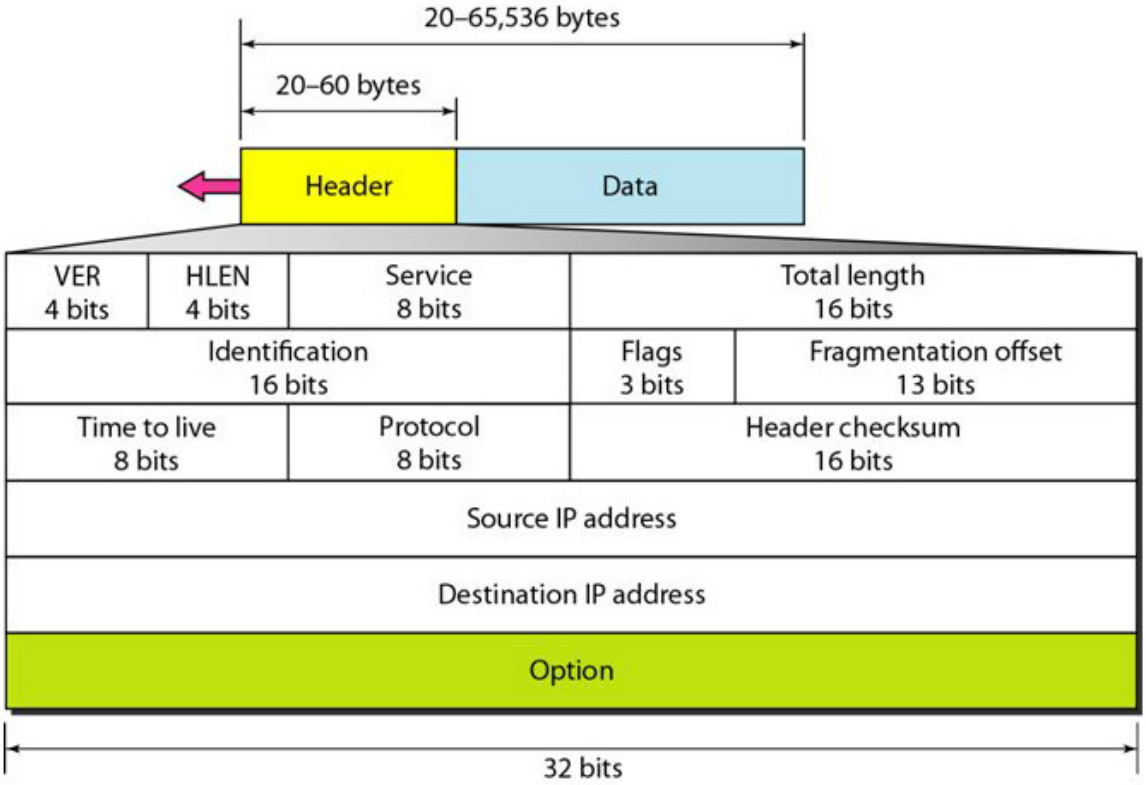
1.2.2 Lớp Internet

Lớp tiếp theo trong một mạng và là lớp đầu tiên ta cần phải quan tâm đến đó là *lớp Internet*. Trong mô hình OSI, lớp Internet có tên gọi là lớp mạng. Một giao thức cho lớp mạng quy định cách các bit và các byte dữ liệu được tổ chức thành một nhóm lớn hơn được gọi là các packet và mô hình đánh địa chỉ để các máy tính có thể tìm thấy nhau trong mạng. *Giao thức Internet* (Internet Protocol - IP) là giao thức được sử dụng rộng rãi nhất trên thế giới và Java hiểu được các giao thức của lớp này.

Giao thức IP có hai phiên bản là IPv4 và IPv6. IPv4 sử dụng các địa chỉ 32 bit và IPv6 sử dụng các địa chỉ 128 bit. IPv6 bổ sung thêm một số tính năng kỹ thuật để trợ giúp quá trình định tuyến. Hiện nay IPv6 đã được sử dụng rộng rãi và có khả năng sẽ vượt qua IPv4 về số lượng người dùng. Cần phải có các gateway đặc biệt và các giao thức đường ống (tunnel) để hai giao thức IPv4 và IPv6 có thể cùng hoạt động trên một mạng.

Cả hai giao thức IPv4 và IPv6 gửi dữ liệu qua lớp Internet trong các packet được gọi là *datagram*. Mỗi một datagram trong IPv4 chứa một phần đầu (header) có độ dài từ 20 byte đến 60 byte và một khối lượng dữ liệu (payload) lên đến 65.515 byte. Trong thực tế thì payload trong IPv4 nhỏ hơn nhiều, từ khoảng vài chục byte đến khoảng 8 kilobyte. Một datagram của IPv6 chứa một header lớn hơn và payload có thể lên đến 4 gigabyte.

Sơ đồ trong *Hình 1.3* trình bày cấu trúc của một IPv4 datagram. Trong sơ đồ này tất cả các bit và các byte được biểu diễn dưới dạng big-endian: MSB đến LSB tính từ trái qua phải.



Hình 1.3: Cấu trúc của một IPv4 datagram

Bên cạnh chức năng định tuyến và đánh địa chỉ, mục đích thứ hai của lớp Internet là cho phép các lớp máy tính đến mạng với nhiều kiểu khác nhau có thể trao đổi được với nhau. Các router Internet chuyển đổi các giao thức giữa WiFi và Ethernet, Ethernet và DSL, DSL và cáp quang. Nếu không có lớp Internet, mỗi máy tính chỉ có thể trao đổi với các máy tính khác trên cùng một kiểu mạng. Lớp Internet có nhiệm vụ kết nối các mạng không đồng nhất sử dụng các giao thức không đồng nhất với nhau.

1.2.3 Lớp giao vận

Các datagram chưa được xử lý có nhiều hạn chế. Đáng chú ý nhất là không có một sự bảo đảm nào cho các datagram sẽ được chuyển giao tới bên nhận, thậm chí nếu được chuyển đi thì các datagram cũng có thể bị hư hỏng trong quá trình truyền. Phần kiểm tra tổng của phần header (header checksum) cũng chỉ có thể phát hiện những hư hỏng trong phần header và không thể phát hiện những hư hỏng trong phần dữ liệu của một datagram. Cuối cùng là các datagram có thể không bị hư hỏng trong quá trình gửi thì cũng có thể sẽ đến đích không theo đúng thứ tự mà các datagram đã được gửi đi vì các datagram có thể đi theo những tuyến đường khác nhau từ nguồn đến đích. Ví dụ: datagram A được gửi trước datagram B nhưng không có nghĩa là datagram A sẽ đến đích trước datagram B.

Lớp giao vận có nhiệm vụ để đảm bảo các gói tin được nhận theo đúng thứ tự đã được gửi đi và dữ liệu không bị mất hay bị hư hỏng. Nếu một gói tin bị mất, lớp giao vận có thể yêu cầu bên gửi gửi lại gói tin đó. Các mạng IP cài đặt cơ chế này bằng cách bổ sung thêm một phần header vào mỗi datagram. Có hai giao thức quan trọng nhất tại lớp giao vận đó là giao thức TCP (Transmission Control Protocol) và giao thức UDP (User Datagram Protocol).

Giao thức TCP là giao thức cho phép truyền lại các dữ liệu bị mất hay bị hư hỏng và chuyển giao các byte theo đúng thứ tự đã được gửi đi. TCP được gọi là giao thức đáng tin cậy. Giao thức UDP cho phép bên nhận phát hiện các gói tin bị hư hỏng nhưng không đảm bảo các gói tin được chuyển giao theo đúng thứ tự. UDP thường nhanh hơn rất nhiều so với TCP. Giao thức UDP là giao thức không tin cậy, tuy nhiên UDP vẫn được sử dụng nhiều trong các mạng Internet.

1.2.4 Lớp ứng dụng

Lớp ứng dụng có trách nhiệm chuyển dữ liệu đến người sử dụng. Ba lớp bên dưới làm việc cùng với nhau để quy định cách dữ liệu được truyền từ một máy tính đến một máy tính khác. Lớp ứng dụng quyết định cần phải làm gì với dữ liệu sau khi dữ liệu đã được truyền tới lớp này. Tồn tại nhiều giao thức khác nhau trong lớp ứng dụng, ví dụ: các giao thức HTTP, HTTPS cho World Wide Web; các giao thức SMTP, POP, IMAP cho email; các giao thức FTP, FSP, TFTP cho truyền file; giao thức NFS cho truy nhập file; các giao thức Gnutella and BitTorrent cho chia sẻ file; các giao thức Session Initiation Protocol (SIP) and Skype cho truyền tiếng nói…

1.3 Giao thức IP, TCP và UDP

1.3.1 Khái quát về giao thức IP, TCP và UDP

Giao thức Internet - IP được phát triển trong thời chiến tranh lạnh giữa Liên Xô và Mỹ và được đỡ đầu bởi quân đội Mỹ. IP cần phải đáp ứng được các yêu cầu:

Đầu tiên, giao thức IP phải đủ mạnh để toàn bộ hệ thống mạng không bị ngừng hoạt động nếu Liên Xô tấn công hạt nhân vào một vị trí nào đó, chẳng hạn ở Cleveland. Tất cả các thông báo vẫn được chuyển đến đích (ngoại trừ đến Cleveland). Do đó, giao thức IP được thiết kế để cho phép có nhiều tuyến đường giữa hai điểm bất kỳ và để định tuyến các gói tin không đi qua các router đã bị hỏng.

Thứ hai, quân đội Mỹ có nhiều loại máy tính khác nhau và tất cả các máy tính này phải trao đổi được với nhau. Do đó, giao thức IP cần phải là một giao thức mở và không phụ thuộc vào hệ điều hành. Do có nhiều tuyến đường giữa hai điểm bất kỳ và đường đi nhanh nhất giữa hai điểm có thể thay đổi liên tục nên các gói tin của

một luồng dữ liệu có thể không đi cùng một tuyến đường đến đích và có thể đến đích không theo đúng thứ tự đã được gửi.

Để cải thiện mục tiêu trên, giao thức TCP đã được đặt bên trên IP để cho phép điểm cuối của một kết nối có khả năng phản hồi các gói tin đã nhận được và yêu cầu gửi lại các gói tin bị mất hoặc bị hư hỏng. TCP còn cho phép các gói tin được sắp xếp lại theo đúng thứ tự đã được gửi đi. Tuy nhiên, TCP lại chứa một phần không nhỏ thông tin phụ được gọi là overhead, phần này sẽ ảnh hưởng đến tốc độ truyền các gói tin. Do đó, nếu thứ tự của các gói tin không thật sự quan trọng và việc một vài gói tin bị hư hỏng hay bị mất không ảnh hưởng đến luồng dữ liệu thì các gói tin đôi khi được gửi đi mà không cần được đảm bảo và TCP cung cấp việc sử dụng giao thức UDP.

Như đã trình bày ở phần *Lớp giao vận*, UDP là giao thức không tin cậy nên giao thức này có thể ảnh hưởng đến việc truyền file. Tuy nhiên giao thức này lại thích hợp cho các ứng dụng khi việc mất một phần dữ liệu sẽ không bị người dùng cuối phát hiện, ví dụ việc mất một vài bit từ một tín hiệu video hay audio sẽ không làm giảm đến chất lượng của dòng bit dữ liệu. Các mã sửa lỗi có thể được đưa vào trong các dòng dữ liệu sử dụng UDP tại mức ứng dụng để xử lý các thông tin bị mất.

Một số lượng lớn các giao thức có thể chạy trên đỉnh của giao thức IP, phổ biến nhất là giao thức Internet Control Message Protocol - ICMP. ICMP sử dụng các datagram chưa được xử lý để chuyển tiếp các thông báo lỗi giữa các máy tính. Ping là chương trình được sử dụng phổ biến nhất trong giao thức ICMP. Java chỉ hỗ trợ TCP, UDP và các giao thức trong lớp ứng dụng, Java không hỗ trợ ICMP. Tất cả các giao thức khác của các lớp giao vận, lớp Internet và các lớp bên dưới chỉ có thể cài đặt trong Java bằng cách triển khai các mã gốc (native code).

1.3.2 Địa chỉ IP và tên miền

Mỗi máy tính trên một mạng IPv4 được xác định bởi một địa chỉ IP duy nhất. Địa chỉ IP này gồm 4 byte, thường được viết dưới dạng bốn chữ số thập phân ngăn cách nhau bởi dấu chấm, ví dụ 192.168.1.1. Mỗi một chữ số thập phân được biểu diễn bằng một byte, do đó phạm vi của mỗi số thập phân sẽ từ 0 đến 255. Khi các gói tin được truyền đi trên mạng, địa chỉ của máy gửi và địa chỉ của máy nhận sẽ được chứa trong phần header của gói tin. Các router dọc theo đường truyền sẽ dựa vào địa chỉ IP của máy nhận để lựa chọn tuyến đường tốt nhất gửi các gói tin đến đích. Máy nhận sẽ sử dụng địa chỉ IP của máy gửi để biết ai đã gửi các gói tin đến mình.

Có khoảng hơn bốn tỷ địa chỉ IPv4, tuy nhiên các địa chỉ này không được phân phối phù hợp giữa các vùng trên thế giới. Tháng 4 năm 2011, châu Á và Australia

sử dụng hết các địa chỉ IP được phân phối. Tháng 9 năm 2012, châu Âu cũng sử dụng hết các địa chỉ IP được phân phối. Bắc Mỹ, Mỹ latinh và châu Phi vẫn còn một số khối địa chỉ IP chưa sử dụng, tuy nhiên các địa chỉ này cũng sẽ nhanh chóng được sử dụng hết.

Có một sự dịch chuyển từ sử dụng IPv4 sang IPv6. IPv6 là các địa chỉ có độ dài 16 byte, đủ để cung cấp địa chỉ IP cho toàn bộ con người, toàn bộ máy tính và trong thực tế toàn bộ các thiết bị trên thế giới. Cách biểu diễn địa chỉ IPv6:

+ Thông thường Các địa chỉ IPv6 được viết thành tám khối, mỗi khối gồm bốn chữ số viết trong hệ thập lục phân, các khối được ngăn cách nhau bởi dấu **:**, ví dụ FEDC:BA98:7654:3210:FEDC:BA98:7654:3210.

+ Các chữ số không (0) ở đầu không cần phải viết. Nếu trong một địa chỉ IPv6 có nhiều khối gồm toàn bộ số 0 thì những khối này có thể được thay thế bằng cặp dấu ::, ví dụ, địa chỉ FEDC:0000:0000:0000:00DC:0000:7076:0010 có thể được viết ngắn gọn thành FEDC::DC:0:7076:10. Mỗi cặp :: chỉ được phép xuất hiện nhiều nhất một lần trong một địa chỉ IPv6

+ Trong các mạng kết hợp cả IPv4 và IPv6, bốn byte cuối cùng của một địa chỉ IPv6 đôi khi được viết như là một địa chỉ IPv4, ví dụ địa chỉ FEDC:BA98:7654:3210:FEDC:BA98:7654:3210 có thể được viết thành FEDC:BA98:7654:3210:FEDC:BA98:118.84.50.16.

Rất khó để con người nhớ được các địa chỉ IP viết dưới dạng các con số, đặc biệt là với địa chỉ IPv6. Do đó, hệ thống tên miền *Domain Name System* (DNS) đã được phát triển để chuyển các địa chỉ IP thành các tên máy (hostname) cho dễ nhớ, ví dụ địa chỉ 208.201.239.101 được chuyển thành *www.oreilly.com*. Khi các chương trình viết bằng Java truy nhập mạng, các chương trình này cần phải xử lý cả các địa chỉ viết bằng các số và cả các hostname tương ứng. Các phương thức để thực hiện công việc này được cung cấp bởi lớp `java.net.InetAddress`.

Một vài máy tính, đặc biệt là các máy chủ, có địa chỉ cố định. Các máy tính khác, đặc biệt là các máy khách trên các mạng cục bộ và các kết nối không dây sẽ nhận các địa chỉ khác nhau mỗi khi các máy tính được khởi động và kết nối vào mạng. Các địa chỉ này được cung cấp bởi DHCP Server. DHCP là viết tắt của Dynamic Host Configuration Protocol - Giao thức cấu hình host động.

Trong địa chỉ IPv4, một vài khối địa chỉ và khuân mẫu có dạng đặc biệt. Tất cả các khối địa chỉ bắt đầu bằng 10., bằng 172.16. đến 172.31. và bằng 192.168. là chưa được cấp cho bất kỳ máy tính nào trên mạng. Các khối địa chỉ này được gọi là các khối địa chỉ không thể định tuyến được (non-routable) và có thể được sử dụng trên

các mạng cục bộ. Không một máy tính nào sử dụng địa chỉ trong các khối trên có thể truy nhập được vào Internet. Các địa chỉ không thể định tuyến được rất thích hợp cho việc xây dựng các mạng riêng, các mạng này sẽ không xuất hiện trên mạng Internet. Các địa chỉ IPv4 bắt đầu bằng 127 luôn có nghĩa là các địa chỉ loopback cục bộ (local loopback address), nghĩa là các địa chỉ này luôn chỉ tới máy tính cục bộ. Hostname cho địa chỉ này thường là localhost. Trong IPv6, địa chỉ 0:0:0:0:0:0:0:1 (cũng được viết tắt là ::1) là địa chỉ loopback. Địa chỉ 0.0.0.0 luôn được chỉ đến máy gửi nhưng cũng có thể được dùng như là địa chỉ nguồn mà không phải địa chỉ đích. Tương tự, trong IPv4, bất kỳ địa chỉ nào bắt đầu bằng 0. (tám bit không) là chỉ đến một máy tính trong cùng một mạng cục bộ.

Trong IPv4 địa chỉ 255.255.255.255 được sử dụng làm địa chỉ quảng bá (broadcast address). Các gói tin được gửi đến địa chỉ này sẽ được tất cả các máy trong mạng cục bộ nhận và sẽ không được gửi ra bên ngoài mạng cục bộ. Địa chỉ quảng bá thường được sử dụng để phát hiện các máy trong cùng một mạng cục bộ, ví dụ: Khi một máy tính trong mạng cục bộ được khởi động, máy tính này sẽ gửi một thông báo đặc biệt đến địa chỉ 255.255.255.255 để tìm DHCP Server. Tất cả các máy tính trong mạng sẽ nhận được thông báo này nhưng chỉ có DHCP Server sẽ trả lời cho máy gửi. Thực tế, DHCP Server gửi sẽ các thông tin về cấu hình mạng cục bộ, bao gồm cả địa chỉ IP mà máy tính vừa khởi động sẽ sử dụng trong phiên làm việc và địa chỉ của DNS Server để máy tính này có thể sử dụng để phân giải tên miền.

1.3.3 Các cổng

Các máy tính hiện đại thực hiện nhiều việc khác nhau cùng một lúc. Email cần phải được tách ra khỏi các yêu cầu FTP, các yêu cầu về FTP cần được tách biệt với lưu lượng truy nhập web. Điều này được thực hiện thông qua cổng (port). Mỗi máy tính với một địa chỉ IP có hàng nghìn cổng logic, chính xác là 65.535 cổng trên giao thức tầng giao vận. Các cổng này hoàn toàn là trừu tượng trong bộ nhớ của máy tính và không phải là các cổng vật lý như cổng USB. Mỗi một cổng được xác định bằng một con số nằm giữa 1 và 65.535. Mỗi một cổng có thể được cấp cho một dịch vụ cụ thể. Ví dụ, giao thức HTTP thường sử dụng cổng 80, chúng ta thường nói máy chủ web lắng nghe (listen) các kết nối đang đến trên cổng 80. Khi một máy tính với một địa chỉ IP cụ thể gửi dữ liệu (thông thường là các yêu cầu) đến máy chủ web thì máy tính cũng sẽ gửi dữ liệu tới cổng 80 trên máy này. Máy chủ web sẽ kiểm tra mỗi gói tin nhận được để phát hiện ra số hiệu của cổng bên máy gửi và sau đó sẽ gửi dữ liệu đến bất kỳ chương trình nào đang nghe trên cổng đó. Bảng sau sẽ liệt kê danh sách các cổng thường được sử dụng trong các ứng dụng.

Giao thức	Cổng	Giao thức	Mục đích
echo	7	TCP/UDP	Giao thức kiểm tra dùng để xác minh hai máy tính có thể kết nối với nhau bằng cách: một máy tính sẽ gửi lại thông tin được gửi từ máy tính phía bên kia.
daytime	13	TCP/UDP	Cung cấp biểu diễn ASCII về thời gian hiện tại trên Server.
ftp data	20	TCP	Được sử dụng để truyền các file.
ftp data	21	TCP	Được sử dụng để gửi các lệnh của FTP như *put* và *get.*
ssh	22	TCP	Được sử dụng để mã hóa đăng nhập an toàn từ xa
telnet	23	TCP	Được sử dụng cho tương tác, các phiên làm việc từ xa.
smtp	25	TCP	Simple Mail Transfer Protocol - Giao thức gửi thư giữa hai máy.
whois	43	TCP	Một dịch vụ về thư mục đơn giản cho các quản trị mạng Internet.
finger	79	TCP	Một dịch vụ trả về thông tin người dùng hay một nhóm người dùng trên hệ thống cục bộ.
http	80	TCP	Giao thức truyền siêu văn bản.
pop3	110	TCP	Port Office Protocol Version 3 - Là giao thức truyền các email được tích lũy cho một khách hàng thỉnh thoảng mới truy nhập mạng .

1.4 Mạng Internet

1.4.1 Các khối địa chỉ Internet

Mỗi nhà cung cấp dịch vụ Internet (ISP) được cấp một số khối địa chỉ IPv4. Khi một tổ chức muốn thiết lập một mạng máy tính kết nối với Internet, ISP sẽ cấp cho tổ chức này một khối địa chỉ, mỗi một khối sẽ được cố định phần đầu của địa chỉ, ví dụ nếu phần đầu của địa chỉ là 216.245.85 thì tổ chức này có thể sử dụng các địa chỉ từ 216.245.85.0 đến 216.245.85.255 (lưu ý các địa chỉ 216.245.85.0 và 216.245.85.255 không được dùng để gán địa chỉ cho các máy tính). Vì phần cố định gồm 24 bit đầu tiên nên sẽ đực ký hiệu là /24. Nếu một khối có phần cố định là /23 thì sẽ còn 9 bit để gán địa chỉ cho các máy tính, do đó sẽ có 2^9-2 = 510 địa chỉ được gán cho các máy tính. Nếu một mạng có có phần cố định là /30 thì chỉ còn 2 bit để gán địa chỉ cho các máy tính và như vậy sẽ chỉ có tối đa 2 địa chỉ được dùng để gán cho các máy tính trong mạng.

1.4.2 Dịch địa chỉ mạng

Do sự khan hiếm địa chỉ IPv4, ngày nay hầu hết các mạng sử dụng giao thức dịch địa chỉ mạng - Network Address Translation (NAT). Trong các mạng dựa trên NAT, hầu hết các nút trong mạng chỉ có địa chỉ cục bộ, không định tuyến được. Các địa chỉ này được lựa chọn tùy ý trong các khối địa chỉ 10.x.x.x, 172.16.x.x. đến 172.31.x.x. và 192.168.x.x. Các router kết nối các mạng cục bộ với ISP sẽ dịch các địa chỉ cục bộ này thành một tập nhỏ hơn các địa chỉ có thể định tuyến được.

1.4.3 Tường lửa

Ngày nay các nguy cơ về an toàn và an ninh mạng đang trở nên hiện hữu, do đó cần phải thiết lập một điểm truy nhập tới mạng cục bộ để kiểm tra tất cả luồng dữ liệu ra và vào mạng tại điểm truy nhập này. Các thiết bị phần cứng và phần mềm được đặt giữa mạng Internet và mạng cục bộ để kiểm tra tất cả các dữ liệu đi vào mạng hay đi ra khỏi mạng cục bộ được gọi là tường lửa (firewall). Firewall sẽ kiểm tra các gói tin và chấp nhận cho các gói tin được đi qua hay sẽ bị hủy bỏ dựa trên một tập hợp các quy định đã được thiết lập. Firewall thường là một bộ phận của router nhưng cũng có thể là là một máy tính chuyên dụng. Các hệ điều hành hiện đại như macOS và Red Hat Linux thường có các firewall cá nhân gắn liền trông hệ điều hành để theo dõi các luồng dữ liệu được gửi đến máy tính này.

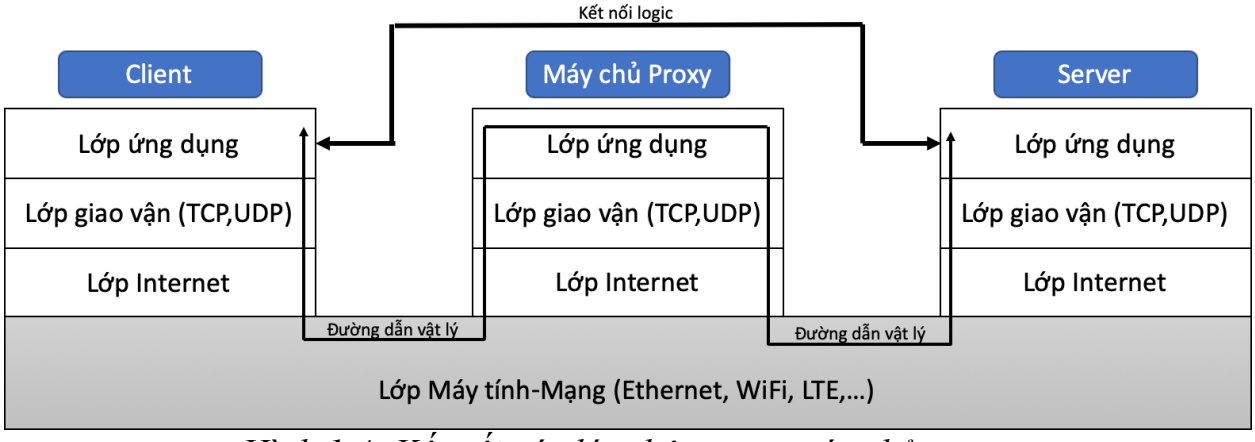
Việc lọc các gói tin thường được dựa trên các địa chỉ mạng và các cổng, ví dụ, tất cả các gói tin đến từ lớp địa chỉ 192.28.25.x có thể bị firewall của một mạng loại bỏ do trước đó một máy tính sử dụng một trong các địa chỉ thuộc lớp này đã tấn công vào mạng.

1.4.4 Máy chủ proxy

Máy chủ *proxy* liên quan đến firewall. Nếu một firewall ngăn cản không cho một máy tính trong mạng kết nối trực tiếp đến các mạng bên ngoài thì máy chủ proxy có thể hoạt động như là trung gian giữa máy tính trong mạng và firewall. Khi một máy tính trong mạng muốn tải một trang từ máy chủ web bên ngoài mạng, máy tính này có thể gửi yêu cầu kết nối đến máy chủ proxy. Máy chủ proxy sau đó có thể yêu cầu trang này từ máy chủ web bên ngoài và chuyển trang đến máy tính đã yêu cầu. Các proxy cũng có thể sử dụng cho các dịch vụ FTP và các kết nối khác. Một trong những ưu điểm về an toàn khi sử dụng máy chủ proxy đó là các máy tính bên ngoài mạng chỉ có thể tìm thấy máy chủ proxy mà không thể tìm được các tên và các địa chỉ IP của các máy bên trong mạng. Điều này sẽ làm cho hacker khó tấn công vào mạng.

Firewall thường hoạt động ở lớp giao vận và lớp Internet, các máy chủ proxy thường hoạt động ở lớp ứng dụng. Một máy chủ proxy hiểu rõ một vài giao thức mức ứng dụng, chẳng hạn như HTTP và FTP. Tuy nhiên các máy chủ proxy SOCKS hoạt động tại lớp giao vận và có thể ủy quyền cho tất cả các kết nối TCP và UDP mà không cần quan tâm đến giao thức được sử dụng ở lớp ứng dụng. Các gói tin có thể được máy chủ proxy kiểm tra để đảm bảo rằng các gói tin này chứa các dữ liệu phù hợp với kiểu của các gói tin. Ví dụ, các gói tin của giao thức FTP chứa các dữ liệu Telnet có thể bị loại bỏ.

Sơ đồ sau trình bày cách kết nối các lớp thông qua máy chủ proxy.



Hình 1.4: Kết nối các lớp thông qua máy chủ proxy

Máy chủ proxy cho phép vừa chuyển tiếp vừa kiểm soát chặt chẽ các truy nhập Internet. Một công ty có thể sử dụng máy chủ proxy để ngăn chặn các truy nhập của người sử dụng trong công ty đến các trang web có nội dung độc hại nhưng lại cho phép những truy nhập đến các trang web có nội dung tốt. Một vài công ty cho phép người sử dụng sử dụng giao thức FTP để nhận các file nhưng không cho phép sử dụng giao thức này để truyền các file ra ngoài công ty, do đó các dữ liệu có tính chất bí mật của công ty không dễ dàng bị mua bán bất hợp pháp. Các công ty có thể sử dụng máy chủ proxy để theo dõi việc truy nhập các trang web của các nhân viên để biết ai sử dụng Internet để hỗ trợ cho công việc, ai sử dụng Internet cho các mục đích khác…

Các máy chủ proxy cũng có thể được sử dụng để cài đặt các vùng nhớ cache cục bộ. Khi người sử dụng yêu cầu một file từ máy chủ web, máy chủ proxy đầu tiên kiểm tra xem file được yêu cầu có trong bộ nhớ cache hay không. Nếu file đã có trong bộ nhớ cache thì máy chủ proxy sẽ gửi file này đến người sử dụng mà không cần phải tìm kiếm trên Internet. Nếu file được yêu cầu không có trong bộ nhớ cache, máy chủ proxy sẽ tìm file này trên Internet và chuyển tiếp đến người sử dụng, đồng thời lưu trữ file đó vào bộ nhớ cache để phục vụ cho các yêu cầu về sau. Cơ chế hoạt

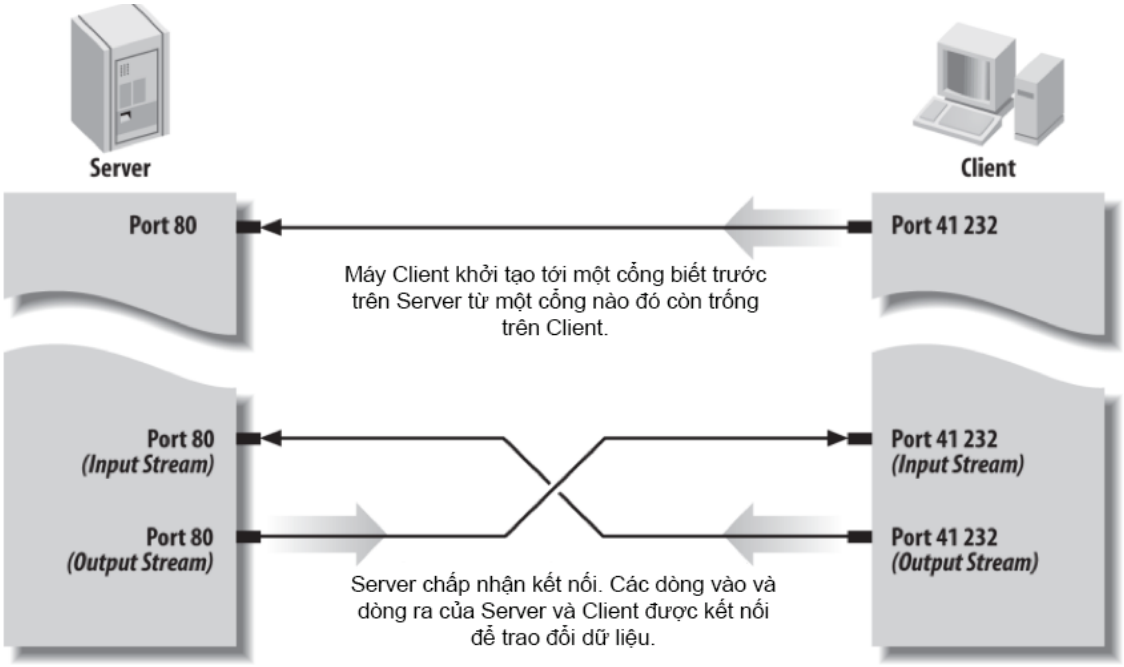
động của bộ nhớ cache sẽ giảm đáng kể truy nhập Internet và cải thiện được thời gian đáp ứng yêu cầu. Một trong các ví dụ điển hình của việc sử dụng các máy chủ proxy như là bộ nhớ cache chính là America Online (AOL).

Bên cạnh những ưu điểm thì máy chủ proxy cũng bộc lộ những hạn chế. Một trong những hạn chế lớn nhất của máy chủ proxy là không có khả năng xử lý với những giao thức mới. Các máy chủ proxy cho phép các giao thức phổ biến đã được thiết lập như HTTP, FTP và SMTP được truyền qua nhưng lại không cho phép các giao thức mới, ví dụ giao thức BitTorrent được truyền qua các máy chủ proxy. Với những thay đổi nhanh chóng của Internet, điều này đã trở thành một trong những hạn chế quan trọng. Trong Java, người lập trình rất dễ dàng tạo ra các giao thức mới để tối ưu hóa công việc. Những hạn chế của máy chủ proxy đã làm giảm tính hiệu quả của các giao thức do người dùng tạo ra vì máy chủ proxy không hiểu được các giao thức này.

1.4.5 Mô hình Client/Server

Hầu hết các lập trình mạng hiện đại đều dựa trên mô hình khách/chủ (Client/Server). Một ứng dụng Client/Server thường lưu trữ các khối lượng lớn dữ liệu trên một hay một chuỗi (còn được gọi là đám mây) các máy chủ có công suất lớn và đắt tiền trong khi hầu hết các chương trình và các giao diện người sử dụng được quản lý bởi phần mềm bên phía Client. Phần mềm này chạy trên các máy tính cá nhân tương đối rẻ tiền. Trong hầu hết các trường hợp, một Server chủ yếu là gửi dữ liệu và Client chủ yếu là nhận dữ liệu đã được gửi đi từ Server, tuy nhiên rất ít chương trình được viết chỉ để gửi hoặc chỉ để nhận dữ liệu. Thông thường, một Client sẽ bắt đầu một cuộc hội thoại và một Server sẽ đợi để bắt đầu cuộc hội thoại với Client. Trong một số trường hợp, cùng một chương trình có thể vừa là Client vừa là Server.

Không phải tất cả các ứng dụng đều thích hợp với mô hình Client/Server. Bên cạnh mô hình Client/Server ta còn có các mô hình kết nối khác chẳng hạn như kết nối ngang hàng (peer to peer). Hệ thống điện thoại là một ví dụ điển hình về mạng kết nối ngang hàng. Mỗi một điện thoại có thể gọi đến các điện thoại khác và cũng có thể nhận các cuộc gọi từ các điện thoại khác nữa. Ngoài ra, các mô hình lai (hybrid) giữa mô hình Client/Server và mô hình Peer-to-Peer cũng được sử dụng rộng rãi trong các ứng dụng mạng.



Hình 1.5: Kết nối Client/Server

Java mặc định không hỗ trợ kết nối ngang hàng trong API. Tuy nhiên, các ứng dụng có thể dễ dàng cung cấp các kết nối ngang hàng theo nhiều cách, phổ biến nhất là chúng hoạt động như là cả Server và cả Client. Một lựa chọn khác là các máy ngang hàng có thể trao đổi với nhau thông qua một chương trình Server trung gian, chương trình này chuyển tiếp các dữ liệu giữa các máy ngang hàng.

CÂU HỎI, BÀI TẬP VẬN DỤNG:

1. Hãy trình bày hiểu biết cơ bản của anh/chị về mạng máy tính?
2. Mô hình TCP/IP có mấy lớp? Anh/chị hãy trình bày hiểu biết của mình về các lớp này?
3. Trình bày khái quát về các giao thức IP, TCP và UDP?
4. Hãy trình bày các khái niệm về địa chỉ IP, tên miền và cổng trong mạng máy tính?
5. Hãy phân biệt tường lửa (firewall) và máy chủ proxy?
6. Mô hình Client/Server là gì? Ngoài mô hình này các ứng dụng mạng còn có thể sử dụng các mô hình nào?

CHƯƠNG 2. CÁC DÒNG VÀO-RA (STREAM)

Các chương trình mạng thường thực hiện các công việc nhập, xuất (I/O) đơn giản đó là chuyển các byte từ một hệ thống này đến một hệ thống khác. Trong Java, I/O được tổ chức tương đối khác so với các ngôn ngữ lập trình khác, chẳng hạn như Fortran, C và C++.

I/O trong Java được xây dựng trong các stream (các dòng). Input stream đọc dữ liệu và ouput stream ghi dữ liệu. Các lớp stream khác nhau như *java.io.FileInputStream* và *sun.net.TelnetOutput* đọc và ghi các nguồn dữ liệu cụ thể. Tuy nhiên tất cả các output stream cùng có các phương thức chung để ghi dữ liệu và tất cả các input stream sử dụng các phương thức chung để đọc dữ liệu.

2.1 Các dòng ra (output stream)

Lớp output cơ bản của Java là *java.io.OutputStream*, với khai báo như sau:

```
public abstract class OutputStream{
}
```

Lớp này cung cấp các phương thức cơ bản để ghi dữ liệu, đó là:

- *public abstract void* ***write****(int b) throws IOException*
- *public void* ***write****(byte[] data) throws IOException*
- *public void* ***write****(byte[] data, int offset, int length) throws IOException*
- *public void* ***flush****() throws IOException*
- *public void* ***close****() throws IOException*

Các lớp con của *OutputStream* sử dụng các phương thức này để ghi dữ liệu lên một phương tiện cụ thể. Ví dụ, một *FileOutputStream* sử dụng các phương thức này để ghi dữ liệu vào một file. Một *TelnetOutputStream* sử dụng các phương thức để ghi dữ liệu vào một kết nối mạng. Một *ByteArrayOutputStream* dùng các phương thức để ghi dữ liệu vào một mảng các byte có thể mở rộng được.

Phương thức cơ bản của *OutputStream* là *write(int b)*. Phương thức này sử dụng một số nguyên có giá trị từ 0 đến 255 như là một đối số và ghi byte tương ứng vào output stream. Phương thức này được mô tả là trừu tượng (abstract) do các lớp con cần thay đổi phương thức này để kiểm soát phương tiện cụ thể. Một *ByteArrayOutputStream* có thể cài đặt phương thức với mã Java để sao chép byte vào mảng trong Java. Tuy nhiên, một *FileOutputStream* sẽ cần phải sử dụng mã riêng, mã này biết cách ghi dữ liệu trong các file trên nền của hệ điều hành.

Mặc dù phương thức *write(int b)* sử dụng một số nguyên như là một đối số nhưng khi ghi sẽ ghi một byte không dấu (unsigned byte). Java không có kiểu dữ liệu byte không dấu nên phải sử dụng một số nguyên. Sự khác biệt giữa một byte không dấu và byte có dấu chính là cách diễn giải. Các byte không dấu và byte có dấu đều được tạo ra từ 8 bit. Khi ta ghi một số nguyên vào một kết nối mạng sử dụng *write(int b)* thì chỉ có 8 bit được đặt lên các đường truyền. Nếu một số nguyên ngoài phạm vi 0 - 255 được truyền cho phương thức *write(int b)*, byte ít quan trọng hơn của số nguyên này sẽ được viết và 3 byte còn lại của số nguyên sẽ bị bỏ qua. Đây chính là hiệu ứng của việc chuyển một số nguyên thành byte.

Ví dụ: Giao thức bộ sinh ký tự (character-generator protocol) định nghĩa một Server gửi ra các văn bản mã hóa bằng mã ASCII. Một trong các biến thể của giao thức này sẽ gửi các dòng, mỗi dòng chứa 72 ký tự mã hóa bằng mã ASCII có thể in ra được. Các ký tự này có số thứ tự từ 33 đến 126 trong bảng mã ASCII ngoại trừ các ký tự trắng và các ký tự điều khiển.

- Dòng đầu chứa các ký tự từ 33 đến 104.
- Dòng thứ hai chứa các ký tự từ 34 đến 105.
- Dòng thứ ba chứa các ký tự từ 35 đến 106.
- Dòng thứ 29 chứa các ký tự từ 55 đến 126.

Đến đây, các ký tự được gói xoay vòng sao cho dòng 30 chứa các ký tự từ 56 đến 126 và sau đó lại là ký tự 33.

Các dòng được kết thúc bằng dấu Enter (ASCII 13) và một mã xuống dòng (ASCII 10).

Đoạn mã sau sẽ trình bày cách triển khai phương thức *write()*:

```
public static void generateCharacters(OutputStream out) throws
IOException {
     int firstPrintableCharacter = 33;
     int numberOfPrintableCharacters = 94;
     int numberOfCharactersPerLine = 72;
     int start = firstPrintableCharacter;
     while (true) { /* infinite loop */
          for (int i = start; i < start + numberOfCharactersPerLine; i++)
          {
               out.write(((i - firstPrintableCharacter) %
numberOfPrintableCharacters) + firstPrintableCharacter);
          }
          out.write('\r'); // carriage return
          out.write('\n'); // linefeed
          start = ((start + 1) - firstPrintableCharacter) %
numberOfPrintableCharacters + firstPrintableCharacter;
     }
}
```

Kết quả của giao thức bộ sinh ký tự là:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefgh
"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghi
#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghij
$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijk
%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijkl
&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklm
'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmn
```

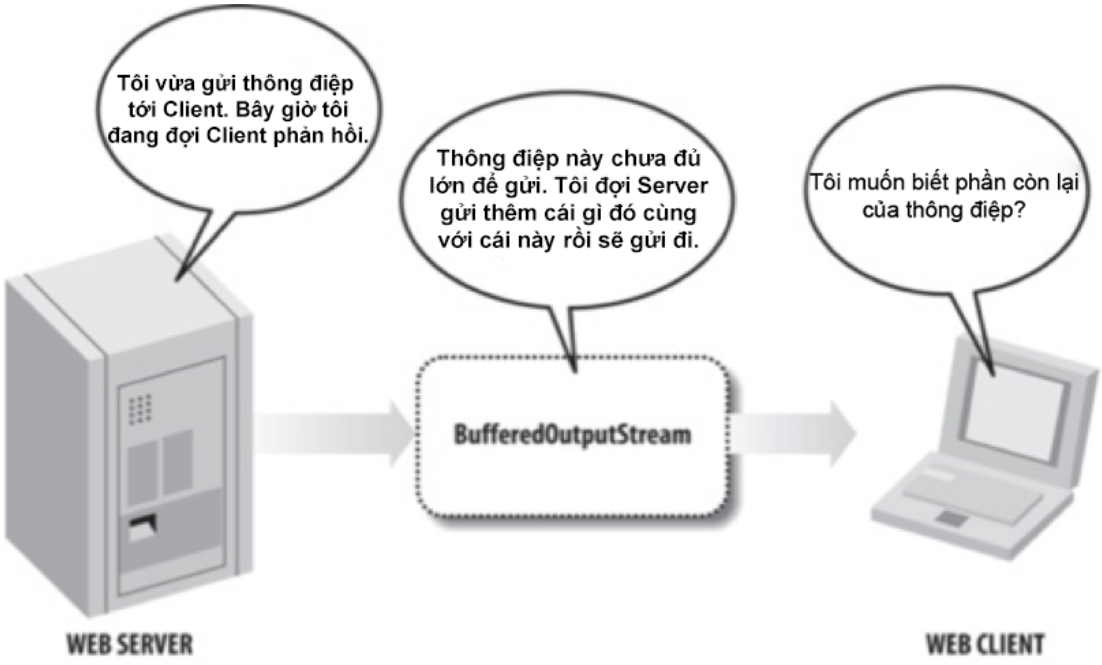
Trong ví dụ trên, một *OutputStream* được truyền đến phương thức *generateCharacters()* trong đối số *out*. Dữ liệu được viết lần lượt từng byte trên *out*. Các byte này được cho dưới dạng các số nguyên trong một chuỗi xoay vòng từ 33 đến 126. Sau mỗi một chuỗi 72 ký tự được viết dấu Enter (ASCII 13) và một mã xuống dòng (ASCII 10) được viết lên output stream. Ký tự kế tiếp được tính toán và vòng lặp sẽ được lặp lại. Phương thức out sử dụng throw *IOException*. Điều này rất quan trọng do Server bộ sinh ký tự sẽ chỉ kết thúc khi bên phía Client kết thúc kết nối và mã Java sẽ nhận biết việc kết thúc này như là một *IOException*.

Việc ghi mỗi lần một byte thường không hiệu quả. Ví dụ, mỗi TCP segment chứa ít nhất 40 byte của phần overhead dùng cho việc định tuyến và sửa lỗi. Nếu mỗi lần chỉ gửi một byte ta có thể làm cho mạng phải tải tổng cộng 41 byte. Do đó, hầu hết các cài đặt TCP/IP sử dụng vùng đệm dữ liệu có một kích thước nào đó. Các cài đặt này sẽ tích lũy dữ liệu cần gửi trong vùng đệm (buffer) và chỉ gửi các dữ liệu này đi khi lượng dữ liệu được tích lũy hay thời gian tích lũy đã vượt một ngưỡng cho trước. Sử dụng *write(byte[] data)* hoặc *write(byte[] data, int offset, int length)* thường nhanh hơn việc ghi lần lượt từng thành phần của một mảng dữ liệu. Sau đây là ví dụ của việc cài đặt phương thức *generateCharacters()*, phương thức này gửi mỗi lần một dòng bằng cách đóng gói toàn bộ một dòng trong một mảng các byte:

```
public static void generateCharacters(OutputStream out) throws
IOException {
     int firstPrintableCharacter = 33;
     int numberOfPrintableCharacters = 94;
     int numberOfCharactersPerLine = 72;
     int start = firstPrintableCharacter;
     byte[] line = new byte[numberOfCharactersPerLine + 2];
     // the +2 is for the carriage return and linefeed
     while (true) { /* infinite loop */
          for (int i = start; i < start + numberOfCharactersPerLine; i++)
{
               line[i - start] = (byte) ((i - firstPrintableCharacter)  %
numberOfPrintableCharacters + firstPrintableCharacter);
          }
          line[72] = (byte) '\r'; // carriage return
          line[73] = (byte) '\n'; // line feed
```

```
            out.write(line);
            start = ((start + 1) - firstPrintableCharacter)
            % numberOfPrintableCharacters + firstPrintableCharacter;
        }
    }
```

Các stream cũng có thể được đưa vào vùng đệm bằng phần mềm như mã Java cũng như là trong phần cứng của mạng. Thông thường, việc đưa dữ liệu vào vùng đệm được thực hiện bằng cách gắn một `BufferedOutputStream` hay một `BufferedWriter` vào một stream lớp dưới. Để đưa dữ liệu trong một vùng đệm lên một kết nối ta sử dụng phương thức `flush()`. Phương thức `flush()` sẽ yêu cầu dòng phải gửi dữ liệu đã được đưa vào vùng đệm lên kết nối, ngay cả khi vùng đệm chưa đầy. Ta cần phải đưa hết tất cả các dữ liệu trong vùng đệm của các dòng bằng phương thức `flush()` trước khi ta đóng các dòng này, nếu không thì khi đóng các dòng, dữ liệu trong các vùng đệm sẽ bị mất.



Hình 2.1: Dữ liệu có thể bị mất nếu không flush các luồng

Khi đã làm việc xong với một stream ta có thể đóng stream này bằng cách gọi phương thức `close()`. Phương thức `close()` sẽ giải phóng tất cả các tài nguyên được liên kết với stream, chẳng hạn như các đặc tả tập tin (file handle) hay các cổng. Nếu stream có được từ một kết nối mạng thì khi đóng stream sẽ hủy bỏ kết nối mạng. Khi một output stream đã bị đóng việc ghi thêm dữ liệu vào stream sẽ đưa ra `IOException`. Tuy nhiên một vài kiểu stream vẫn cho phép chúng ta tiếp tục làm việc với đối tượng khi đã đóng stream. Ví dụ, một `ByteArrayOutputStream` đã bị đóng vẫn có thể được chuyển đổi thành mảng các byte thật sự và một `DigestOutputStream` đã bị đóng vẫn có thể trả về giá trị digest của nó.

Nếu không đóng một stream trong một chương trình lớn có thể làm tiết lộ các đặc tả tập tin, các cổng mạng và các tài nguyên khác. Do đó, với Java 6 và các phiên bản trước đó, ta nên có một khối có tên là *finally* để đóng các dòng, khối này là khối cuối cùng trong một chương trình. Thông thường, ta khai báo biến *stream* bên ngoài khối *try* nhưng nên khởi tạo dòng bên trong khối *try*. Để tránh gặp phải *NullPointerExceptions* ta cần kiểm tra xem biến *stream* có phải là *null* hay không trước khi ta đóng dòng này. Ví dụ sau minh họa cách viết sử dụng *finally*:

```
OutputStream out = null;
try {
     out = new FileOutputStream("/tmp/data.txt");
     // work with the output stream...
} catch (IOException ex) {
     System.err.println(ex.getMessage());
} finally {
     if (out != null) {
          try {
               out.close();
          } catch (IOException ex) {
          }
     }
}
```

Kỹ thuật này đôi khi được gọi là *dispose pattern*. Dispose pattern phổ biến cho các đối tượng cần phải được dọn dẹp trước khi các dữ liệu không còn sử dụng được thu dọn. Dispose pattern không chỉ được sử dụng cho các đối tượng và còn được sử dụng cho cả các sockets, channels, JDBC connections và các statements.

Java 7 giới thiệu cách sử dụng khối *try* cùng với xây dựng các tài nguyên để thu dọn dữ liệu không còn sử dụng. Với cách này, dòng được khai báo trong một danh sách đối số bên trong khối *try*. Đoạn mã trên được viết lại như sau:

```
try (OutputStream out = new FileOutputStream("/tmp/data.txt")) {
     // work with the output stream...
} catch (IOException ex) {
     System.err.println(ex.getMessage());
}
```

Trong đoạn mã này ta nhận thấy khối *finally* đã không còn cần thiết. Java sẽ tự động gọi phương thức *close()* trên bất kỳ đối tượng có thuộc tính *AutoCloseable* bên trong danh sách đối số của khối *try*.

2.2 Các dòng vào (input stream)

Lớp input cơ bản của Java là *java.io.InputStream*, với khai báo như sau:

```
public abstract class InputStream{
}
```

Lớp này cung cấp các phương thức cơ bản để đọc dữ liệu như là các byte chưa được xử lý còn được gọi là byte thô:

- *public abstract int* ***read****() throws IOException*
- *public int* ***read****(byte[] input) throws IOException*
- *public int* ***read****(byte[] input, int offset, int length) throws IOException*
- *public long* ***skip****(long n) throws IOException*
- *public int* ***available****() throws IOException*
- *public void* ***close****() throws IOException*

Các lớp con cụ thể của *InputStream* sử dụng các phương thức này để đọc dữ liệu từ một phương tiện cụ thể. Ví dụ, một *FileOutputStream* sử dụng các phương thức này để đọc dữ liệu từ một file. Một *TelnetOutputStream* sử dụng các phương thức này để đọc dữ liệu từ một kết nối mạng. Một *ByteArrayOutputStream* sử dụng các phương thức này để đọc dữ liệu từ một mảng các byte.

Phương thức cơ bản của *InputStream* là phương thức *read()* không có đối số. Phương thức *read()* đọc một byte dữ liệu từ nguồn của một input stream và trả lại một số nguyên nằm trong khoảng từ 0 đến 255. Kết thúc của một stream được báo hiệu bằng cách trả lại giá trị *-1*. Phương thức *read()* sẽ chờ và ngăn không cho thực thi bất kỳ đoạn mã nào sau phương thức này cho đến khi một byte dữ liệu sẵn sàng cho việc đọc.

Việc đọc và ghi dữ liệu có thể chậm. Do đó nếu chương trình đang phải thực thi một đoạn mã quan trọng thì nên đặt các thao tác I/O trong một luồng (thread) riêng của thao tác I/O.

Phương thức *read()* được mô tả là trừu tượng do các lớp con cần thay đổi phương thức này để quản lý các phương tiện cụ thể. Ví dụ, một *ByteArrayInputStream* có thể thực thi phương thức này bằng mã Java để sao chép byte từ một mảng. Tuy nhiên, một *TelnetInputStream* cần sử dụng một thư viện riêng để biết cách đọc dữ liệu từ một giao diện mạng trên nền của hệ điều hành.

Đoạn mã sau đọc 10 byte từ *InputStream* có tên là *in* và lưu các byte trong một mảng byte có tên *input*. Trong quá trình đọc nếu phát hiện đã hết dữ liệu thì vòng lặp sẽ được kết thúc sớm hơn:

```
byte[] input = new byte[10];
for (int i = 0; i < input.length; i++) {
     int b = in.read();
     if (b == -1) break;
     input[i] = (byte) b;
}
```

Mặc dù *read()* chỉ đọc một byte, phương thức này trả lại một số nguyên, nên cần phải thực hiện chuyển đổi một số nguyên thành một byte bằng cast trước khi lưu trữ kết quả trong mảng byte. Vì *read()* trả lại một giá trị nguyên có phạm vi từ -128 đến 127 thay vì 0 đến 255 nên ta cần phải chuyển đổi một byte có dấu thành byte không dấu như sau:

```
int i = b >= 0 ? b : 256 + b;
```

Cũng tương tự như việc ghi mỗi lần một byte trong *OutputStream*, việc đọc mỗi lần một byte sẽ là không hiệu quả. Có hai phương thức overloaded là *read(byte[] input)* và *read(byte[] input, int offset, int length)*. Hai phương thức này sẽ làm đầy một mảng các byte từ một dòng. Phương thức thứ nhất sẽ cố gắng làm đầy một mảng các byte có tên *input*. Phương thức thứ hai sẽ cố gắng làm đầy một mảng con các byte của mảng *input*, bắt đầu từ vị trí *offset* và sẽ ghi *length* byte. Việc ghi vào một mảng đôi khi sẽ không thành công vì nhiều lý do khác nhau. Ví dụ, tại một thời điểm, ta cố gắng để đọc 1024 byte từ một kết nối mạng, khi đó ta có thể nhận được 512 byte từ Server còn 512 byte đang được chuyển đến. Để biết được số lượng byte đã được đọc ta sử dụng các phương thức đọc nhiều byte. Đoạn mã sau trình bày cách đọc nhiều byte:

```
byte[] input = new byte[1024];
int bytesRead = in.read(input);
```

Đoạn mã sẽ cố gắng đọc 1024 byte từ *InputStream* vào một mảng các byte có tên là *input*. Tuy nhiên, nếu chỉ có 512 byte để đọc thì *bytesRead* sẽ được thiết lập là 512.

Để đảm bào đọc được tất cả các byte ta đặt thao tác đọc vào trong một vòng lặp cho đến khi mảng được làm đầy. Ví dụ:

```
int bytesRead = 0;
int bytesToRead = 1024;
byte[] input = new byte[bytesToRead];
while (bytesRead < bytesToRead) {
     bytesRead += in.read(input, bytesRead, bytesToRead - bytesRead);
}
```

Tất cả các phương thức *read()* sẽ trả lại giá trị -1 để báo hiệu kết thúc một stream. Nếu một stream kết thúc trong khi vẫn còn dữ liệu chưa được đọc thì các phương thức đọc nhiều byte sẽ vẫn đọc dữ liệu cho đến khi vùng đệm trở nên trống. Khi đó việc đọc tiếp sẽ trả lại giá trị *-1* và giá trị này sẽ không được ghi vào mảng. Do đó trong mảng chỉ chứa dữ liệu thực sự. Đoạn mã trên chưa cân nhắc đến trường hợp tất cả 1024 byte chưa đến được vùng đệm. Do đó, ta cần kiểm tra giá trị của *read()* trước khi bổ sung giá trị này vào *bytesRead*. Ví dụ:

```
int bytesRead = 0;
int bytesToRead = 1024;
byte[] input = new byte[bytesToRead];
while (bytesRead < bytesToRead) {
     int result = in.read(input, bytesRead, bytesToRead - bytesRead);
     if (result == -1) break; // end of stream
     bytesRead += result;
}
```

Ta có thể sử dụng phương thức *available()* để xác định số lượng byte có thể đọc ngay mà không cần phải chờ. Phương thức này trả lại số lượng byte tối thiểu mà ta có thể đọc. Ví dụ:

```
int bytesAvailable = in.available();
byte[] input = new byte[bytesAvailable];
int bytesRead = in.read(input, 0, bytesAvailable);
// continue with rest of program immediately...
```

Trong một số ít trường hợp nếu muốn bỏ qua việc đọc dữ liệu, ta sử dụng phương thức *skip()*. Khi đã đọc xong dữ liệu từ một stream ta nên đóng stream bằng cách gọi phương thức *close()* của input stream. Phương thức này sẽ giải phóng tất cả các tài nguyên liên kết với dòng như các đặc tả tập tin hay các cổng. Khi một input strream đã bị đóng, các thao tác đọc dữ liệu tiếp theo sẽ đưa ra *IOException*. Tuy nhiên một vài kiểu stream có thể vẫn cho phép làm một số công việc với đối tượng. Ví dụ, ta sẽ không thể nhận được message digest từ một *java.security.DigestInputStream* cho đến khi dữ liệu đã được đọc và dòng bị đóng lại.

Mark và Reset

Lớp *InputStream* có ba phương thức ít được sử dụng, các phương thức này cho phép các chương trình sao chép dự phòng (backup) và đọc lại dữ liệu đã được đọc trước đó:

- *public void* ***mark****(int readAheadLimit)*
- *public void* ***reset****() throws IOException*
- *public boolean* ***markSupported****()*

Để đọc lại dữ liệu cần đánh dấu (mark) vị trí hiện thời trong dòng bằng phương thức *mark()*. Sử dụng phương thức *reset()* để thiết lập lại (reset) dòng tại điểm đã đánh dấu. Các thao tác đọc tiếp theo sẽ trả lại dữ liệu bắt đầu từ điểm đánh dấu. Số lượng các byte có thể đọc từ vị trí đánh dấu được xác định bởi đối số *readAheadLimit* trong *mark()*. Nếu ta quay trở lại vượt quá điểm đã đánh dấu thì chương trình sẽ đưa ra *IOException*. Trong một dòng luôn chỉ tồn tại duy nhất một đánh dấu. Đánh dấu vị trí thứ hai sẽ xóa bỏ đánh dấu thứ nhất.

Đánh dấu và thiết lập lại thường được thực hiện bằng cách lưu trữ các byte được đọc kể từ vị trí đã được đánh dấu trong một vùng đệm bên trong chương trình. Tuy nhiên không phải tất cả các dòng đều hỗ trợ mark và reset.

Để kiểm tra xem một dòng có hỗ trợ mark và reset không ta sử dụng phương thức `markSupported()`. Nếu phương thức này trả lại giá trị là `true` thì dòng có hỗ trợ mark và reset. Nếu phương thức này trả lại giá trị là `false` thì dòng không hỗ trợ mark và reset, khi đó `mark()` sẽ không làm gì và `reset()` sẽ đưa ra một `IOException`.

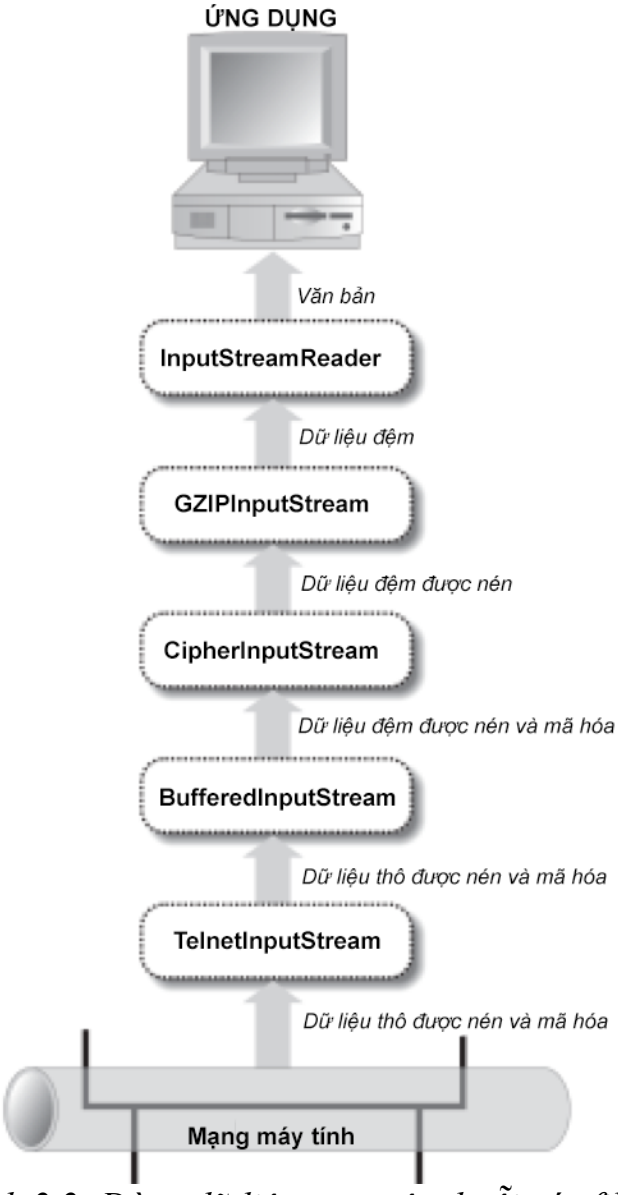
Chỉ có các lớp `BufferedInputStream` và `ByteArrayInputStream` trong `java.io` là luôn hỗ trợ mark và reset. Tuy nhiên các input stream khác chẳng hạn như `TelnetInputStream` cũng có thể hỗ trợ mark nếu các lớp này lần đầu được gắn vào với một `BufferedInputStream`.

2.3 Các dòng filter stream

`InputStream` và `OutputStream` là các lớp tương đối đơn giản. Các lớp này chỉ thực hiện việc đọc và ghi từng byte hay một nhóm các byte. Việc xác định ý nghĩa của các byte, chẳng hạn như các byte có phải là các số nguyên, là các số thực dấu phảy động IEEE 754 hay là văn bản được mã hóa bằng bảng mã Unicode hoàn toàn phụ thuộc vào người lập trình và mã chương trình. Tuy nhiên có một vài định dạng dữ liệu vô cùng phổ biến đã được cài đặt trong thư viện lớp. Ví dụ, rất nhiều các số nguyên được truyền đi như là các phần của các giao thức mạng là các số nguyên 32 bit big-endian; nhiều văn bản được truyền trên web được mã hóa hoặc là ASCII 7-bit, Latin-1 8-bit, hay UTF-8 multibyte; nhiều file được truyền đi bằng giao thức FTP được lưu trữ dưới định dạng ZIP. Java cung cấp một số lượng các lớp lọc (filter classes) có thể được gắn vào các dòng để dịch các byte thô thành các định dạnh khác.

Các filter được chia thành hai kiểu: các filter stream reader và writer. Filter stream chủ yếu làm việc với các dữ liệu thô như các byte, chẳng hạn như nén dữ liệu hoặc diễn dịch dữ liệu là các số nhị phân. Reader và writer quản lý trường hợp đặc biệt của văn bản trong nhiều cách mã hóa khác nhau như UTF-8 và ISO 8859-1.

Các filter được tổ chức trong một chuỗi (chain). Mỗi liên kết trong chuỗi nhận dữ liệu từ filter trước hoặc từ dòng và truyền dữ liệu đến liên kết kế tiếp trong chuỗi. Sơ đồ sau trình bày chuỗi các filter.



Hình 2.2: Dòng dữ liệu qua một chuỗi các filter

Trong ví dụ trên, một file văn bản đã được mã hóa và được nén được truyền đến thông qua giao diện mạng cục bộ. Mã riêng của chương trình chuyển file này đến `TelnetInputStream`. Một `BufferedInputStream` sẽ đưa file vào vùng đệm để tăng tốc toàn bộ quá trình. Một `CipherInputStream` giải mã dữ liệu và một `GZIPInputStream` sẽ giải nén dữ liệu. Một `InputStreamReader` chuyển đổi dữ liệu đã được giải nén thành file văn bản mã hóa bằng mã Unicode. Cuối cùng, file văn bản đã sẵn sảng cho ứng dụng trong lớp ứng dụng và được xử lý.

Các filter output stream có cùng các phương thức như `write()`, `close()` và `flush()` như `java.io.OutputStream`. Các filter input stream có cùng các phương thức như `read()`, `close()` và `available()` như `java.io.InputStream`.

2.3.1 Gắn kết các filter stream

Các filter được kết nối với các dòng bằng các constructor của filter. Đoạn mã sau sẽ đưa vào vùng đệm dữ liệu từ file *data.txt*. Đầu tiên, đối tượng *fin* của *FileInputStream* được tạo bằng cách truyền tên của file như là một đối số cho constructor của *FileInputStream*. Sau đó một đối tượng có tên *bin* của *BufferedInputStream* được tạo bằng cách truyền *fin* như là một đối số cho constructor của *BufferedInputStream*:

```
FileInputStream fin = new FileInputStream("data.txt");
BufferedInputStream bin = new BufferedInputStream(fin);
```

Từ đây ta có thể sử dụng phương thức *read()* của cả *fin* và *bin* để đọc dữ liệu từ file *data.txt*. Trong phần lớn thời gian, ta chỉ nên sử dụng filter cuối cùng trong chuỗi để thực hiện việc đọc và ghi thực sự.

2.3.2 Các lớp BufferedInputStream và BufferedOutputStream

Lớp *BufferedOutputStream* lưu trữ dữ liệu đã được ghi trong một vùng đệm (một mảng byte được bảo vệ có tên là *buf*) cho đến khi vùng đệm đầy hoặc dòng được đẩy đi. Sau đó lớp này sẽ ghi một lần tất cả dữ liệu vào output stream lớp dưới. Với cùng một số byte cần ghi lên một kết nối mạng, việc ghi một lần nhiều byte sẽ nhanh hơn nhiều việc ghi mỗi lần một byte do mỗi TCP segment hay mỗi gói UDP sẽ phải mang thêm một phần overhead đến 40 byte.

Lớp *BufferedInputStream* cũng có một mảng byte được bảo vệ có tên là *buf*, mảng này phục vụ như là một vùng đệm. Mỗi khi phương thức *read()* của dòng được gọi, đầu tiên phương thức này sẽ cố gắng để tìm đọc các dữ liệu đã được yêu cầu từ vùng đệm. Chỉ khi vùng đệm không còn dữ liệu thì dòng sẽ đọc dữ liệu từ các nguồn lớp dưới. Khi đó dòng sẽ đọc càng nhiều dữ liệu có thể được từ nguồn lớp dưới vào vùng đệm, cho dù dòng có thể cần hay không cần các dữ liệu này ngay tức thì. Các dữ liệu không được sử dụng ngay tức thì sẽ được sử dụng cho các lần gọi phương thức *read()* về sau. Sử dụng vùng đệm có thể cải thiện đáng kể hiệu suất. *BufferedInputStream* và *BufferedOutputStream* đều có hai constructor:

- *public* ***BufferedInputStream****(InputStream in)*
- *public* ***BufferedInputStream****(InputStream in, int bufferSize)*
- *public* ***BufferedOutputStream****(OutputStream out)*
- *public* ***BufferedOutputStream****(OutputStream out, int bufferSize)*

Đối số thứ nhất là dòng lớp dưới mà từ dòng này các dữ liệu chưa được đưa vào vùng đệm sẽ được đọc hoặc các dữ liệu đã được đưa vào vùng đệm sẽ được ghi lên dòng lớp dưới này. Đối số thứ hai (nếu có) sẽ xác định số lượng byte trong vùng đệm, nếu không thì kích thước của vùng đệm sẽ được thiết lập bằng 2018 byte cho một input stream và bằng 512 byte cho một output stream. Kích thước lý tưởng cho một vùng đệm phụ thuộc vào kiểu của dòng mà ta đang đưa vào vùng đệm.

`BufferedInputStream` không có các phương thức riêng mà chỉ ghi đè (override) lên các phương thức của `InputStream`. `BufferedInputStream` hỗ trợ mark và reset.

`BufferedOutputStream` cũng không có các phương thức riêng. Ta có thể gọi các phương thức của `BufferedOutputStream` giống như cách ta gọi các phương thức của bất kỳ output stream nào. Điểu khác biệt ở đây là phương thức `write()` trong `BufferedOutputStream` sẽ đặt dữ liệu trong một vùng đệm thay vì ghi trực tiếp xuống output stream lớp dưới. Do đó ta cần phải thực hiện đẩy dữ liệu ra khỏi vùng đệm bằng phương thức `flush()` tại thời điểm mà dữ liệu cần phải gửi đi.

2.3.3 Lớp PrintStream

Lớp `PrintStream` là filter output stream đầu tiên mà hầu hết các lập trình viên gặp do `System.out` là một `PrintStream`. Tuy nhiên nhiều output stream khác cũng được gắn vào print stream, sử dụng 2 constructor:

- `public PrintStream(OutputStream out)`
- `public PrintStream(OutputStream out, boolean autoFlush)`

Mặc định, các print stream cần phải được đẩy ra kết nối, nếu đối số `autoFlush` là `true` thì dòng sẽ được đẩy ra kết nối mỗi khi một mảng byte hoặc ký hiệu `linefeed` được viết hoặc một phương thức `println()` được gọi.

`PrintStream` có 9 phương thức overloaded `print()` và 10 phương thức overloaded `println()`:

- `public void print(boolean b)`
- `public void print(char c)`
- `public void print(int i)`
- `public void print(long l)`
- `public void print(float f)`
- `public void print(double d)`
- `public void print(char[] text)`
- `public void print(String s)`
- `public void print(Object o)`

- `public void println()`
- `public void println(boolean b)`
- `public void println(char c)`
- `public void println(int i)`
- `public void println(long l)`
- `public void println(float f)`
- `public void println(double d)`
- `public void println(char[] text)`
- `public void println(String s)`
- `public void println(Object o)`

Mỗi phương thức `print()` chuyển đổi đối số của nó thành một xâu và ghi xâu này trên output stream lớp dưới sử dụng cách mã hóa mặc định. Phương thức `println()` cũng có chung mục đích như phương thức `print()` nhưng có bổ sung ký hiệu ngăn cách dòng (ký hiệu này phụ thuộc vào hệ điều hành) vào cuối của dòng mà phương thức này sẽ ghi. Ký hiệu ngăn cách dòng trong Unix là `\n`, trong Mac OS 9 là `\r`, trong Windows là một cặp (`\r\n`).

2.3.4 Các lớp DataInputStream và DataOutputStream

Các lớp `DataInputStream` và `DataOutputStream` cung cấp các phương thức cho việc đọc và ghi các kiểu dữ liệu cơ bản của Java và các xâu dưới dạng nhị phân. Các định dạng nhị phân được sử dụng cho các mục đích trao đổi dữ liệu giữa các chương trình Java khác nhau thông qua một kết nối mạng, một file dữ liệu, một đường ống (pipe line) hoặc các môi trường trung gian khác

Lớp `DataOutputStream` cung cấp 11 phương thức để ghi các kiểu dữ liệu cụ thể của Java:

- `public final void writeBoolean(boolean b) throws IOException`
- `public final void writeByte(int b) throws IOException`
- `public final void writeShort(int s) throws IOException`
- `public final void writeChar(int c) throws IOException`
- `public final void writeInt(int i) throws IOException`
- `public final void writeLong(long l) throws IOException`
- `public final void writeFloat(float f) throws IOException`
- `public final void writeDouble(double d) throws IOException`
- `public final void writeChars(String s) throws IOException`
- `public final void writeBytes(String s) throws IOException`
- `public final void writeUTF(String s) throws IOException`

Tất cả các dữ liệu được viết dưới dạng big-endian. Các số nguyên được viết dưới dạng bù hai với số lượng byte ít nhất. Do đó, một số nguyên khai báo kiểu `byte` được viết như là 1 byte, khai báo kiểu `short` sẽ được biểu diễn bằng 2 byte, khai báo kiểu `int` sẽ được biểu diễn bằng 4 byte, khai báo kiểu `long` sẽ được biểu diễn bằng 8 byte. Các số thực khai báo kiểu `float` và `double` sẽ được viết dưới dạng IEEE 754 và được biểu diễn lần lượt là 4 và 8 byte. Các dữ liệu kiểu boolean sẽ được biểu diễn bằng 1 bit với giá trị `0` cho `false` và `1` cho `true`. Các ký tự được viết dưới dạng 2 byte không dấu.

Cùng với các phương thức để viết các số nhị phân và các xâu, `DataOutputStream` vẫn có các phương thức thông thường như `write()`, `flush()`, và `close()` như bất kỳ lớp output stream nào.

`DataInputStream` là lớp bổ sung cho lớp `DataOutputStream`. `DataInputStream` có thể đọc tất cả các định dạng mà lớp `DataOutputStream` ghi. Thêm vào đó, lớp `DataInputStream` cũng có các phương thức thông thường như `read()`, `available()`, `skip()` và `close()` cũng như các phương thức để đọc toàn bộ các mảng các byte và các dòng của văn bản.

Có 9 phương thức để đọc đọc dữ liệu nhị phân tương thích với 11 phương thức trong `DataOutputStream`:

- `public final boolean readBoolean() throws IOException`
- `public final byte readByte() throws IOException`
- `public final char readChar() throws IOException`
- `public final short readShort() throws IOException`
- `public final int readInt() throws IOException`
- `public final long readLong() throws IOException`
- `public final float readFloat() throws IOException`
- `public final double readDouble() throws IOException`
- `public final String readUTF() throws IOException`

Lớp `DataInputStream` cung cấp hai phương thức để đọc các byte không dấu và các số nguyên khai báo kiểu short không dấu và trả về số nguyên tương ứng. Mặc dù Java không có hai kiểu dữ liệu là byte không dấu và số nguyên kiểu short không dấu nhưng người lập trình trong Java có thể gặp các kiểu dữ liệu này khi đọc dữ liệu nhị phân viết bằng ngôn ngữ C. Hai phương thức là:

- `public final int readUnsignedByte() throws IOException`
- `public final int readUnsignedShort() throws IOException`

Lớp *DataInputStream* có hai phương thức thông thường để đọc dữ liệu vào môt mảng hoặc một mảng con và trả về số byte đã đọc. Lớp *DataInputStream* cũng có hai phương thức *readFully()*, hai phương thức này sẽ lặp lại việc đọc dữ liệu vào môt mảng từ input stream lớp dưới cho đến khi đã đọc đủ số byte đã yêu cầu. Nếu không đọc được đủ dữ liệu một *IOException* sẽ được đưa ra. Các phương thức là:

- *public final int* ***read****(byte[] input) throws IOException*
- *public final int* ***read****(byte[] input, int offset, int length) throws IOException*
- *public final void* ***readFully****(byte[] input) throws IOException*
- *public final void* ***readFully****(byte[] input, int offset, int length) throws IOException*

DataInputStream cung cấp phương thức phổ biến *readLine()* để đọc một dòng trong văn bản và trả lại một dòng:

- *public final String* ***readLine****() throws IOException*

Tuy nhiên ta không nên sử dụng phương thức này vì phương thức này bị phản đối và có thể gây ra lỗi trong quá trình đọc.

2.4 Các lớp Reader and Writer

2.4.1 Lớp Writer

Lớp *Writer* tương ứng với lớp *java.io.OutputStream*. Lớp *Writer* là *abstract* và có hai *protected constructor*. Cũng giống như *OutputStream*, lớp *Writer* không bao giờ được sử dụng trực tiếp, thay vào đó lớp này được sử dụng theo kiểu polymorphic thông qua một trong số các lớp con của nó. Lớp *Writer* có năm phương thức *write()* và các phương thức *flush()*, *close()*:

- *protected* ***Writer****()*
- *protected* ***Writer****(Object lock)*
- *public abstract void* ***write****(char[] text, int offset, int length) throws IOException*
- *public void* ***write****(int c) throws IOException*
- *public void* ***write****(char[] text) throws IOException*
- *public void* ***write****(String s) throws IOException*
- *public void* ***write****(String s, int offset, int length) throws IOException*
- *public abstract void* ***flush****() throws IOException*

- *public abstract void* ***close****() throws IOException*

Phương thức *write(char[] text, int offset, int length)* là phương thức cơ bản để cho bốn phương thức *write()* còn lại được triển khai. Một lớp con cần phải ít nhất override phương thức này cũng như là override các phương thức *flush()* và *close()*.

2.4.2 Lớp OutputStreamWriter

OutputStreamWriter là lớp con cụ thể quan trọng nhất của lớp *Writer*. Một *OutputStreamWriter* nhận các ký tự từ một chương trình Java sau đó sẽ chuyển đổi thành các byte dựa trên một cách mã hóa cụ thể và ghi các byte này lên một output stream lớp dưới. Constructor của *OutputStreamWriter* xác định output stream để ghi vào và cách mã hóa được sử dụng:

- *public* ***OutputStreamWriter****(OutputStream out, String encoding) throws UnsupportedEncodingException*

Ngoài các constructor, *OutputStreamWriter* chỉ có các phương thức *Writer* thông thường được sử dụng giống như trong lớp *Writer* và một phương thức để trả về mã hóa của đối tượng:

- *public String* ***getEncoding****()*

2.4.3 Lớp Reader

Lớp *Reader* tương ứng với lớp *java.io.InputStream*. Lớp *Reader* là *abstract* và có hai *protected constructor*. Giống như *InputStream* và *Writer*, lớp *Reader* không bao giờ được dùng trực tiếp và chỉ được sử dụng thông qua một trong số các lớp con của nó. Lớp *Reader* có ba phương thức *read()* và các phương thức *skip()*, *close()*, *ready()*, *mark()*, *reset()* và *markSupported()*:

- *protected* ***Reader****()*
- *protected* ***Reader****(Object lock)*
- *public abstract int* ***read****(char[] text, int offset, int length) throws IOException*
- *public int* ***read****() throws IOException*
- *public int* ***read****(char[] text) throws IOException*
- *public long* ***skip****(long n) throws IOException*
- *public boolean* ***ready****()*
- *public boolean* ***markSupported****()*
- *public void* ***mark****(int readAheadLimit) throws IOException*
- *public void* ***reset****() throws IOException*
- *public abstract void* ***close****() throws IOException*

Phương thức *read(char[] text, int offset, int length)* là phương thức cơ bản qua đó hai phương thức *read()* còn lại được cài đặt. Một lớp con cần phải ít nhất override phương thức này cũng như là override phương thức *close()*.

InputStreamReader là lớp con cụ thể quan trọng nhất của *Reader*. Một *InputStreamReader* đọc các byte từ một input stream lớp dưới, chẳng hạn như một *FileInputStream* hay *TelnetInputStream* sau đó chuyển đổi các byte này thành các ký tự dựa trên một cách mã hóa cụ thể và trả về các ký tự này. Constructor của *InputStreamReader* xác định input stream để đọc và cách mã hóa được sử dụng:

- *public* ***InputStreamReader****(InputStream in)*
- *public* ***InputStreamReader****(InputStream in, String encoding) throws UnsupportedEncodingException*

2.4.4 Các lớp Filter Reader và Filter Writer

Các lớp *InputStreamReader* và *OutputStreamWriter* hoạt động trên đỉnh của các input stream và output stream. Các lớp này thay đổi giao diện từ một giao diện hướng byte (byte-oriented interface) thành một giao diện hướng ký tự (character-oriented interface). Sau khi đã thay đổi giao diện, các bộ lọc hướng ký tự (character-oriented filters) bổ sung sẽ được đặt trên đỉnh của reader và writer sử dụng các lớp *java.io.FilterReader* và *java.io.FilterWriter*. Cũng như các filter stream, có nhiều lớp con thực hiện các thao tác lọc cụ thể, bao gồm:

- ***BufferedReader***
- ***BufferedWriter***
- ***LineNumberReader***
- ***PushbackReader***
- ***PrintWriter***

Các lớp *BufferedReader* và *BufferedWriter* là các lớp dựa trên ký tự (character-based) tương đương với các lớp hướng byte (byte-oriented) *BufferedInputStream* và *BufferedOutputStream*. Khi một chương trình đọc dữ liệu từ một *BufferedReader*, văn bản được lấy từ vùng đệm mà không phải được lấy trực tiếp từ input stream lớp dưới hay các nguồn khác. Khi vùng đệm không có dữ liệu, nó sẽ lại được làm đầy với số lượng ký tự càng nhiều càng tốt. Các ký tự này có thể sẽ không được sử dụng ngay mà có thể được sử dụng cho các lần đọc sau. Khi một chương trình ghi vào một *BufferedWriter*, văn bản sẽ được đưa vào vùng đệm. Văn bản chỉ được chuyển đến output stream lớp dưới hay các đích đến khác khi vùng đệm đã đầy hoặc khi vùng đệm được flush, điều này sẽ làm cho việc ghi trở nên nhanh hơn.

`BufferedReader` và `BufferedWriter` có những phương thức thông dụng kết hợp với các reader và writer là `read()`, `ready()`, `write()` và `close()`. Mỗi lớp có hai constructor, các constructor gắn `BufferedReader` hoặc `BufferedWriter` với reader hoặc writer lớp dưới và thiết lập kích thước của vùng đệm. Nếu kích thước của vùng đệm không được thiết lập thì kích thước mặc định của vùng đệm sẽ là 8192 ký tự:

- `public BufferedReader(Reader in, int bufferSize)`
- `public BufferedReader(Reader in)`
- `public BufferedWriter(Writer out)`
- `public BufferedWriter(Writer out, int bufferSize)`

Lớp `BufferedReader` cũng có một phương thức `readLine()` để đọc một dòng văn bản và trả về như một dòng:

- `public String readLine() throws IOException`

Phương thức này thay thế cho phương thức đã bị phản đối `readLine()` trong `DataInputStream`. Sự khác biệt giữa hai phương thức này khi gắn một `BufferedReader` vào một `InputStreamReader` ta có thể đọc chính xác các dòng trong một tập hợp các ký tự thay vì cách mã hóa mặc định của hệ thống.

Lớp `BufferedWriter` bổ sung một phương thức mới không có trong lớp cha (superclass), được gọi là `newLine()`, phương thức này hướng đến việc ghi các dòng:

- `public void newLine() throws IOException`

2.4.5 Lớp Scanner

Lớp `Scanner` trong Java dùng để quét trên các dòng vào (bàn phím, socket, xâu kí tự…) để lấy được các giá trị mong muốn: xâu kí tự, số nguyên, số thực,…

Một số phương thức thường được sử dụng của lớp `Scanner` trong Java:

Phương thức	Mô tả
`public String next()`	Trả về một xâu kí tự trước khoảng trắng
`public String nextLine()`	Trả về kết quả nội dung của một dòng
`public byte nextByte()`	Trả về kiểu dữ liệu byte
`public short nextShort()`	Trả về kiểu dữ liệu short
`public int nextInt()`	Trả về kiểu dữ liệu int
`public long nextLong()`	Trả về kiểu dữ liệu long
`public float nextFloat()`	Trả về kiểu dữ liệu float
`public double nextDouble()`	Trả về kiểu dữ liệu double

Đoạn mã sau cho phép nhập một số nguyên từ bàn phím và in ra số đó:

```
import java.util.Scanner;
public ViDuScanner {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Vui lòng nhập một: ");
        int so = sc.nextInt();
        System.out.print("Số bạn vừa nhập: " + so);
    }
}
```

2.4.6 Lớp PrintWriter

Lớp *PrintWriter* thay thế cho lớp *PrintStream* trong Java 1.0. Lớp này thích hợp cho việc kiểm soát các ký tự được mã hóa bằng nhiều byte và các văn bản quốc tế.

Ngoài các constructor, lớp *PrintWriter* cũng có một tập hợp các phương thức tương tự như lớp *PrintStream*, đó là:

- *public* ***PrintWriter****(Writer out)*
- *public* ***PrintWriter****(Writer out, boolean autoFlush)*
- *public* ***PrintWriter****(OutputStream out)*
- *public* ***PrintWriter****(OutputStream out, boolean autoFlush)*
- *public void* ***flush****()*
- *public void* ***close****()*
- *public boolean* ***checkError****()*
- *public void* ***write****(int c)*
- *public void* ***write****(char[] text, int offset, int length)*
- *public void* ***write****(char[] text)*
- *public void* ***write****(String s, int offset, int length)*
- *public void* ***write****(String s)*
- *public void* ***print****(boolean b)*
- *public void* ***print****(char c)*
- *public void* ***print****(int i)*
- *public void* ***print****(long l)*
- *public void* ***print****(float f)*
- *public void* ***print****(double d)*
- *public void* ***print****(char[] text)*
- *public void* ***print****(String s)*
- *public void* ***print****(Object o)*
- *public void* ***println****()*
- *public void* ***println****(boolean b)*
- *public void* ***println****(char c)*
- *public void* ***println****(int i)*
- *public void* ***println****(long l)*
- *public void* ***println****(float f)*
- *public void* ***println****(double d)*
- *public void* ***println****(char[] text)*

- `public void println(String s)`
- `public void println(Object o)`

Hầu hết các phương thức trên hoạt động tương tự như các phương thức trong `PrintStream` ngoại trừ bốn phương thức `write()`. Bốn phương thức `write()` sẽ không ghi các byte mà ghi các ký tự.

Trong lập trình mạng hiện nay, chúng ta thường dùng lớp `Scanner` làm input stream và `PrintWriter` làm output stream vì 2 lớp này hỗ trợ tốt việc đọc và ghi nhiều kiểu dữ liệu khác nhau với các phương thức linh hoạt, hiệu quả.

CÂU HỎI, BÀI TẬP VẬN DỤNG:

1. Các lớp *InputStream* và *OutputStream* trong Java được khai báo như thế nào? Liệt kê các phương thức cơ bản của 2 lớp này?
2. Trình bày hiểu biết của anh (chị) về các lớp *BufferedInputStream*, *BufferedOutputStream*?
3. Trình bày cấu tạo của lớp *PrintStream*?
4. Các lớp *DataInputStream* và *DataOutputStream* có những đặc điểm gì cần chú ý?
5. Trình bày cấu tạo của lớp *Reader*, *Writer* và các lớp dẫn xuất của nó?
6. Lớp *Scanner* và *PrintWriter* dùng để làm gì? Nêu cấu tạo của chúng? Tại sao hai lớp này được dùng nhiều trong các ứng dụng mạng?

CHƯƠNG 3. LẬP TRÌNH ĐA LUỒNG TRONG JAVA

3.1 Giới thiệu về luồng (thread)

3.1.1 Thread là gì? Multi-thread là gì?

Thread (luồng) về cơ bản là một tiến trình con (sub-process). Một đơn vị xử lý nhỏ nhất của máy tính có thể thực hiện một công việc riêng biệt. Trong Java, các luồng được quản lý bởi máy ảo Java (JVM).

Multi-thread (đa luồng) là một tiến trình thực hiện nhiều luồng đồng thời. Một ứng dụng Java ngoài luồng chính có thể có các luồng khác thực thi đồng thời làm ứng dụng chạy nhanh và hiệu quả hơn.

Trình duyệt web hay các chương trình chơi nhạc là ví dụ điển hình về đa luồng:

+ Khi duyệt 1 trang web, có rất nhiều hình ảnh, mã CSS, mã JavaScript… được tải đồng thời bởi các luồng khác nhau.

+ Khi đang chơi nhạc, chúng ta vẫn có thể tương tác được với nút điều khiển như: play, pause, next, back … vì luồng phát nhạc là luồng riêng biệt với luồng tiếp nhận tương tác của người dùng.

3.1.2 Đa nhiệm (multitasking)

Multitasking là khả năng chạy đồng thời một hoặc nhiều chương trình cùng một lúc trên một hệ điều hành. Hệ điều hành quản lý việc này và sắp xếp lịch phù hợp cho các chương trình đó. Ví dụ, trên hệ điều hành Windows chúng ta có làm việc đồng thời với các chương trình khác nhau như: Word, Excel, Chrome, …

Chúng ta sử dụng đa nhiệm để tận dụng tính năng của CPU. Đa nhiệm có thể đạt được bằng hai cách:

a) Đa nhiệm dựa trên tiến trình (process) – Đa tiến trình (multiprocessing).

- Mỗi tiến trình có địa chỉ riêng trong bộ nhớ, tức là mỗi tiến trình được phân bổ vùng nhớ riêng biệt.
- Đa nhiệm dựa trên tiến trình là *nặng*.
- Sự giao tiếp giữa các tiến trình có *chi phí cao*.
- Chuyển đổi từ tiến trình này sang tiến trình khác đòi hỏi thời gian để đăng ký việc lưu và tải các bản đồ bộ nhớ, các danh sách cập nhật, …

b) Đa nhiệm dựa trên luồng (thread) – Đa luồng (multithreading).

- Các luồng chia sẻ không gian địa chỉ ô nhớ giống nhau.
- Đa nhiệm dựa trên luồng là *nhẹ*.

- Sự giao tiếp giữa các luồng có *chi phí thấp*.

Đa tiến trình (multiprocessing) và đa luồng (multithreading) đều được sử dụng để tạo ra hệ thống đa nhiệm (multitasking). Nhưng chúng ta sử dụng đa luồng nhiều hơn đa tiến trình bởi vì các luồng chia sẻ một vùng bộ nhớ chung. Chúng không phân bổ vùng bộ nhớ riêng biệt để tiết kiệm bộ nhớ, và chuyển đổi ngữ cảnh giữa các luồng mất ít thời gian hơn giữa các tiến trình.

3.1.3 Ưu điểm và nhược của đa luồng

Ưu điểm:

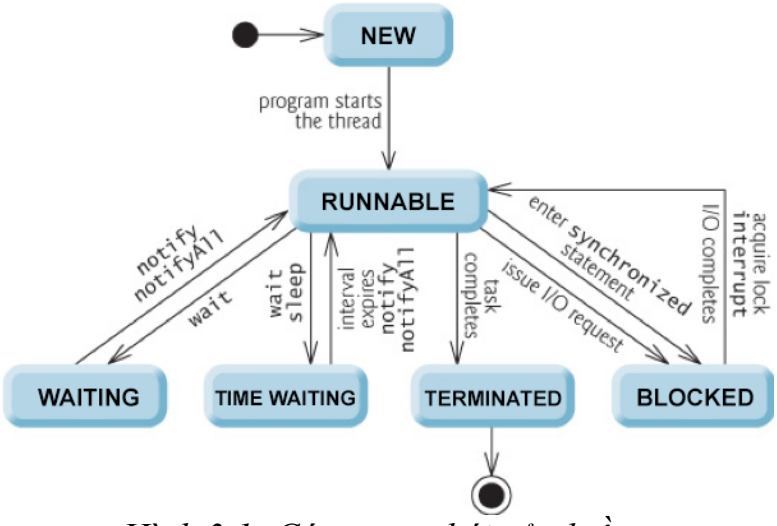
- Đa luồng không chặn người sử dụng vì các luồng là độc lập và chúng ta có thể thực hiện nhiều công việc cùng một lúc.
- Mỗi luồng có thể dùng chung và chia sẻ nguồn tài nguyên trong quá trình chạy, nhưng có thể thực hiện một cách độc lập.
- Luồng là độc lập vì vậy nó không ảnh hưởng đến luồng khác nếu ngoại lệ xảy ra trong một luồng duy nhất.
- Chương trình có thể thực hiện nhiều hoạt động với nhau để tiết kiệm thời gian. Ví dụ một ứng dụng có thể được tách thành: luồng chính chạy giao diện người dùng và các luồng phụ gửi kết quả xử lý đến luồng chính.

Nhược điểm:

- Càng nhiều luồng thì xử lý càng phức tạp.
- Xử lý vấn đề về tranh chấp bộ nhớ, đồng bộ dữ liệu khá phức tạp.
- Cần phát hiện tránh các luồng chết (deadlock) - luồng chạy mà không làm gì trong ứng dụng cả. Deadlock xảy ra khi 2 tiến trình đợi nhau hoàn thành trước khi chạy, kết quả của quá trình là cả 2 tiến trình không bao giờ kết thúc.

3.2 Vòng đời của một luồng trong Java

Vòng đời của luồng trong Java được kiểm soát bởi JVM. Java định nghĩa các trạng thái của luồng trong các thuộc tính `static` của lớp `Thread.State`:



Hình 3.1: Các trạng thái của luồng

NEW: Đây là trạng thái khi luồng vừa được khởi tạo bằng phương thức khởi tạo của lớp `Thread` nhưng chưa được `start()`. Ở trạng thái này, luồng được tạo ra nhưng chưa được cấp phát tài nguyên và cũng chưa chạy. Nếu luồng đang ở trạng thái này mà ta gọi các phương thức ép buộc `stop()`, `resume()`, `suspend()`… sẽ là nguyên nhân xảy ra ngoại lệ `IllegalThreadStateException`.

RUNNABLE: Sau khi gọi phương thức `start()` thì luồng đã được cấp phát tài nguyên và các lịch điều phối CPU cho luồng cũng bắt đầu có hiệu lực. Ở đây, chúng ta dùng trạng thái là *Runnable* chứ không phải Running, vì luồng không thực sự luôn chạy mà tùy vào hệ thống mà có sự điều phối CPU khác nhau.

BLOCKED: Đây là 1 dạng của trạng thái *Not Runnable*. Luồng chờ được unlock mới hoạt động trở lại.

TERMINATED: Một luồng ở trong trạng thái terminated hoặc dead khi phương thức `run()` của nó bị thoát.

TIMED WAITING: Luồng chờ trong một thời gian nhất định, hoặc là có một luồng khác đánh thức nó.

WAITING: Luồng chờ không giới hạn cho đến khi được một luồng khác đánh thức nó.

3.3 Cách tạo luồng trong Java

Trong Java ta có thể tạo ra một luồng bằng một trong hai cách sau: tạo 1 đối tượng của lớp được kế thừa từ lớp `Thread` hoặc hiện thực từ giao diện `Runnable`.

3.3.1 Tạo luồng bằng cách kế thừa từ lớp Thread

Để tạo luồng bằng cách tạo lớp kế thừa từ lớp *Thread*, ta phải làm các công việc sau :

1. Khai báo 1 lớp mới kế thừa từ lớp *Thread*.
2. Override lại phương thức *run()* ở lớp này, những gì trong phương thức *run()* sẽ được thực thi khi luồng bắt đầu chạy. Sau khi luồng chạy xong tất cả các câu lệnh trong phương thức *run()* thì luồng cũng tự hủy.
3. Tạo 1 instance (hay 1 đối tượng) của lớp vừa khai báo.
4. Sau đó gọi phương thức *start()* của đối tượng này để bắt đầu thực thi luồng.

Ví dụ đơn giản về tạo luồng từ lớp *Thread*:

```
package vn.tbit.simple;

public class TheadSimple extends Thread {
    public void run() {
        System.out.println("Thread đang chạy...");
    }
    public static void main(String args[]) {
        TheadSimple t1 = new TheadSimple();
        t1.start();
    }
}
```

Lưu ý :

- Tuy khai báo những công việc cần làm của luồng trong phương thức *run()* nhưng khi thực thi luồng ta phải gọi phương thức *start()*. Vì đây là phương thức đặc biệt mà Java xây dựng sẵn trong lớp *Thread*, phương thức này sẽ cấp phát tài nguyên cho luồng mới rồi chạy phương thức *run()* ở luồng này. Vì vậy, nếu ta gọi phương thức *run()* mà không gọi *start()* thì cũng tương đương với việc gọi một phương thức của một đối tượng bình thường và phương thức vẫn chạy trên luồng đã gọi phương thức chứ không chạy ở luồng mới tạo ra. Như vậy, vẫn chỉ có một luồng chính làm việc chứ ứng dụng vẫn không phải là đa luồng.
- Sau khi gọi *start()* một luồng thì không bao giờ có thể gọi *start()* lại. Nếu làm như vậy, một ngoại lệ *IllegalThreadStateException* sẽ xảy ra.

3.3.2 Tạo luồng bằng cách hiện thực từ giao diện Runnable

Để tạo luồng bằng cách hiện thực từ *Interface Runnable*, ta phải làm các công việc sau :

1. Khai báo 1 lớp mới *implements* từ *Interface Runnable*.
2. Hiện thực phương thức *run()* ở lớp này, những gì trong phương thức *run()* sẽ được thực thi khi luồng bắt đầu chạy. Sau khi luồng chạy xong tất cả các câu lệnh trong phương thức *run()* thì luồng cũng tự hủy.
3. Tạo 1 instance (hay 1 đối tượng) của lớp vừa khai báo, giả sử là *r1*.
4. Tạo 1 instance của lớp *Thread* bằng phương thức khởi tạo *Thread(Runnable target)* trong đó *target* là 1 đối tượng thuộc lớp được *implements* từ giao diện *Runnable*.

   Ví dụ: *Thread t1 = new Thread(r1);*
5. Gọi phương thức *start()* của đối tượng *t1*.

Ví dụ đơn giản về tạo luồng từ giao diện *Runnable*:

```
package vn.tbit.simple;

public class RunnableSimple implements Runnable {
    public void run() {
        System.out.println("Thread đang chạy...");
    }

    public static void main(String args[]) {
        RunnableSimple runable = new RunnableSimple();
        Thread t1 = new Thread(runable);
        t1.start();
    }
}
```

Khi nào *implements* từ *Interface Runnable*?

+ Cách hay được sử dụng và được yêu thích là dùng *Interface Runnable*, bởi vì nó không yêu cầu phải tạo một lớp kế thừa từ lớp *Thread*. Trong trường hợp ứng dụng thiết kế yêu cầu sử dụng đa kế thừa, chỉ có interface mới có thể giúp giải quyết vấn đề. Ngoài ra nó cũng rất hiệu quả và được cài đặt, sử dụng rất đơn giản.

+ Trong trường hợp còn lại ta có thể kế thừa từ lớp *Thread*.

3.4 Ví dụ minh họa sử dụng đa luồng

***Ví dụ 3-1**. Tạo luồng bằng cách extends từ lớp Thread.*

Tạo lớp *extends* từ *Thread*:

```
package vn.tbit.flow;

public class ThreadDemo extends Thread {
    private Thread t;
    private String threadName;
```

```
    ThreadDemo(String name) {
        threadName = name;
        System.out.println("Creating " + threadName);
    }

    @Override
    public void run() {
        System.out.println("Running " + threadName);
        try {
            for (int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", " + i);
                // Thread tạm nghỉ.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
        System.out.println("Thread " + threadName + " exiting.");
    }

    public void start() {
        System.out.println("Starting " + threadName);
        if (t == null) {
            t = new Thread(this, threadName);
            t.start();
        }
    }
}
```

Chương trình sử dụng đa luồng:

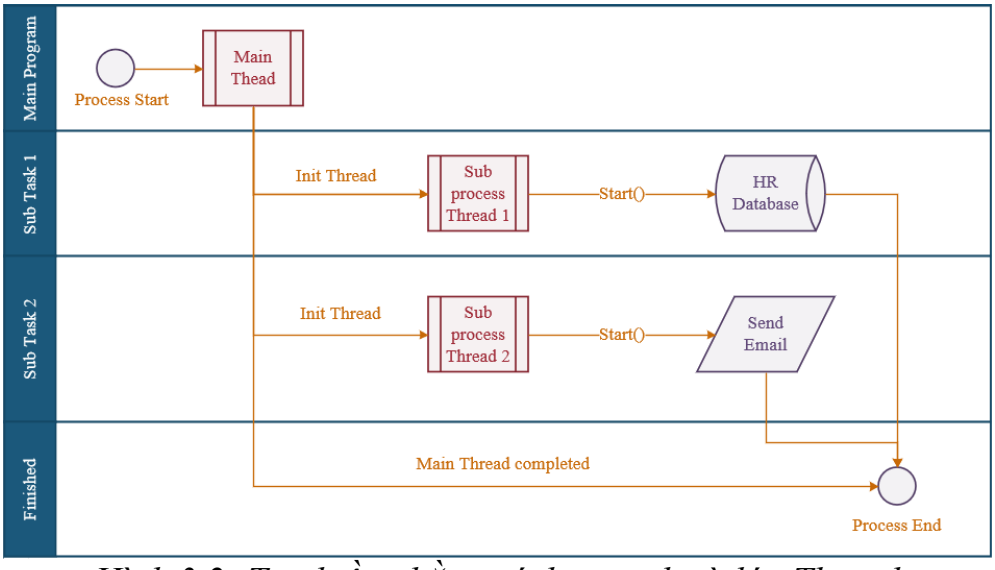
```
package vn.tbit.flow;

public class ThreadDemoTest {
    public static void main(String args[]) {
        System.out.println("Main thread running... ");

        ThreadDemo T1 = new ThreadDemo("Thread-1-HR-Database");
        T1.start();

        ThreadDemo T2 = new ThreadDemo("Thread-2-Send-Email");
        T2.start();

        System.out.println("==> Main thread stopped!!! ");
    }
}
```

Kết quả thực thi chương trình trên:

```
Main thread running...
Creating Thread-1-HR-Database
Starting Thread-1-HR-Database
Creating Thread-2-Send-Email
Starting Thread-2-Send-Email
==> Main thread stopped!!!
```

```
Running Thread-1-HR-Database
Running Thread-2-Send-Email
Thread: Thread-2-Send-Email, 4
Thread: Thread-1-HR-Database, 4
Thread: Thread-1-HR-Database, 3
Thread: Thread-2-Send-Email, 3
Thread: Thread-2-Send-Email, 2
Thread: Thread-1-HR-Database, 2
Thread: Thread-2-Send-Email, 1
Thread: Thread-1-HR-Database, 1
Thread Thread-2-Send-Email exiting.
Thread Thread-1-HR-Database exiting.
```

Kết quả chương trình trên được giải thích thông qua hình bên dưới:



Hình 3.2: Tạo luồng bằng cách extends từ lớp Thread

***Ví dụ 3-2**. Tạo luồng bằng cách `implements` từ giao diện `Runnable`.*

Tạo lớp `implements` từ giao diện `Runnable`:

```
package vn.tbit.flow;

class RunnableDemo implements Runnable {
    private Thread t;
    private String threadName;

    RunnableDemo(String name) {
        threadName = name;
        System.out.println("Creating " + threadName);
    }

    @Override
    public void run() {
        System.out.println("Running " + threadName);
        try {
            for (int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", " + i);
                // Let the thread sleep for a while.
```

```
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
        System.out.println("Thread " + threadName + " exiting.");
    }

    public void start() {
        System.out.println("Starting " + threadName);
        if (t == null) {
            t = new Thread(this, threadName);
            t.start();
        }
    }
}
```

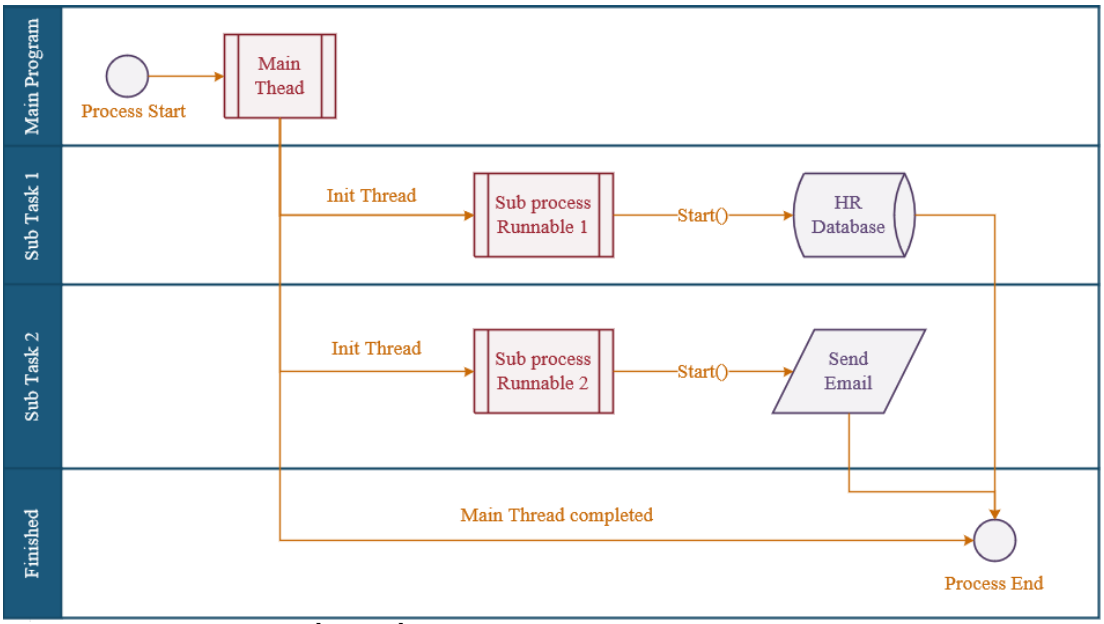
Chương trình sử dụng đa luồng:

```
package vn.tbit.flow;

public class RunnableDemoTest {
    public static void main(String args[]) {
        System.out.println("Main thread running... ");

        RunnableDemo R1 = new RunnableDemo("Thread-1-HR-Database");
        R1.start();

        RunnableDemo R2 = new RunnableDemo("Thread-2-Send-Email");
        R2.start();

        System.out.println("==> Main thread stopped!!! ");
    }
}
```

Kết quả thực thi chương trình trên:

```
Main thread running...
Creating Thread-1-HR-Database
Starting Thread-1-HR-Database
Creating Thread-2-Send-Email
Starting Thread-2-Send-Email
==> Main thread stopped!!!
Running Thread-1-HR-Database
Running Thread-2-Send-Email
Thread: Thread-1-HR-Database, 4
Thread: Thread-2-Send-Email, 4
Thread: Thread-1-HR-Database, 3
Thread: Thread-2-Send-Email, 3
Thread: Thread-1-HR-Database, 2
Thread: Thread-2-Send-Email, 2
Thread: Thread-1-HR-Database, 1
Thread: Thread-2-Send-Email, 1
Thread Thread-1-HR-Database exiting.
Thread Thread-2-Send-Email exiting.
```

Kết quả chương trình trên được giải thích thông qua hình bên dưới:



Hình 3.3: Tạo luồng bằng cách implements từ giao diện Runnable

3.5 Các phương thức của lớp Thread thường hay sử dụng

- `suspend()`: phương thức làm tạm dừng hoạt động của 1 luồng nào đó bằng các ngưng cung cấp CPU cho luồng này. Để cung cấp lại CPU cho luồng ta sử dụng phương thức `resume()`. Cần lưu ý là ta không thể dừng ngay hoạt động của luồng bằng phương thức này. Phương thức `suspend()` không dừng ngay tức thì hoạt động của luồng mà sau khi luồng này trả CPU về cho hệ điều hành thì không cấp CPU cho luồng nữa.
- `resume()`: phương thức làm cho luồng chạy lại khi luồng bị dừng do phương thức `suspend()` bên trên. Phương thức này sẽ đưa luồng vào lại lịch điều phối CPU để luồng được cấp CPU chạy lại bình thường.
- `stop()`: phương thức này sẽ kết thúc phương thức `run()` bằng cách ném ra 1 ngoại lệ `ThreadDeath`, điều này cũng sẽ làm luồng kết thúc 1 cách ép buộc. Nếu giả sử, trước khi gọi `stop()` mà luồng đang nắm giữa 1 đối tượng nào đó hoặc 1 tài nguyên nào đó mà luồng khác đang chờ thì có thể dẫn tới việc xảy ra deadlock.
- `destroy()`: dừng hẳn luồng.
- `isAlive()`: phương thức này kiểm tra xem luồng còn *active* hay không. Phương thức sẽ trả về `true` nếu luồng đã được `start()` và chưa rơi vào trạng thái *dead*. Nếu phương thức trả về `false` thì luồng đang ở trạng thái *New Thread* hoặc là đang ở trạng thái *dead*.
- `yeild()` : hệ điều hành đa nhiệm sẽ phân phối CPU cho các tiến trình, các luồng theo vòng xoay. Mỗi luồng sẽ được cấp CPU trong 1 khoảng thời gian nhất

định, sau đó trả lại CPU cho hệ điều hành, hệ điều hành sẽ cấp CPU cho luồng khác. Khi gọi phương thức này luồng sẽ bị ngừng cấp CPU và nhường cho luồng tiếp theo trong hàng chờ *Ready*. Luồng không phải ngưng cấp CPU như `suspend()` mà chỉ ngưng cấp trong lần nhận CPU đó mà thôi.

- `sleep(long)`: tạm dừng luồng trong một khoảng thời gian tính bằng mili giây.
- `join()`: thông báo rằng hãy chờ luồng này hoàn thành rồi luồng cha mới được tiếp tục chạy.
- `join(long)`: luồng cha cần phải đợi sau mili giây mới được tiếp tục chạy, kể từ lúc gọi `join(long)`. Nếu tham số bằng `0` nghĩa là đợi cho tới khi luồng này kết thúc.
- `getName()`: trả về tên của luồng.
- `setName(String name)`: thay đổi tên của luồng.
- `getId()`: trả về id của luồng.
- `getState()`: trả về trạng thái của luồng.
- `currentThread()`: trả về tham chiếu của luồngđang được thi hành.
- `getPriority()`: trả về mức độ ưu tiên của luồng.
- `setPriority(int)`: thay đổi mức độ ưu tiên của luồng.
- `isDaemon()`: kiểm tra nếu luồnglà một luồng Daemon.
- `setDaemon(boolean)`: thiết lập luồnglà một luồng Daemon hay không.
- `interrupt()`: làm gián đoạn một luồng trong Java. Nếu luồng nằm trong trạng thái sleep hoặc wait, nghĩa là `sleep()` hoặc `wait()` được gọi ra. Việc gọi phương thức `interrupt()` trên luồng đó sẽ phá vỡ trạng thái sleep hoặc wait và ném ra ngoại lệ `InterruptedException`. Nếu luồng không ở trong trạng thái sleep hoặc wait, việc gọi phương thức `interrupt()` thực hiện hành vi bình thường và không làm gián đoạn luồng nhưng đặt cờ interrupt thành `true`.
- `isInterrupted()`: kiểm tra luồng nào đó đã bị ngắt hay không.
- `interrupted()`: kiểm tra xem luồng hiện tại đã bị ngắt hay không.

3.6 Một số vấn đề liên quan đến luồng

3.6.1 Một số tham số của luồng

Định danh của luồng (ThreadId): *ThreadId* là định danh của luồng, được dùng để phân biệt với các luồng khác cùng tiến trình hoặc cùng tập luồng. Đây là thông số mà máy ảo Java tự tạo ra khi ta tạo luồng nên ta không thể sửa đổi cũng

như áp đặt thông số này khi tạo luồng. Nhưng ta có thể lấy được *ThreadId* thông qua phương thức `getId()` của lớp `Thread`.

Tên của luồng (ThreadName): *ThreadName* là tên của luồng, đây là thuộc tính mà ta có thể đặt hoặc không đặt cho luồng. Nếu ta không đặt cho luồng thì máy ảo Java sẽ tự đặt với quy tắc sau: "Thread-" + Thứ tự luồng được tạo ra, bắt đầu từ 0.

Độ ưu tiên của luồng (Priority): Như đã nói ở phần trước, mỗi luồng có 1 độ ưu tiên nhất định. Đây sẽ là thông số quyết định mức ưu tiên khi cấp phát CPU cho các luồng.

Trong Java, để đặt độ ưu tiên cho 1 luồng ta dùng phương thức: `void setPriority(int newPriority)`

- `int newPriority`: Mức độ ưu tiên từ 1 đến 10.

Java có định nghĩa sẵn 3 mức ưu tiên chuẩn như sau:

- `Thread.MIN_PRIORITY` (giá trị 01)
- `Thread.NORM_PRIORITY` (giá trị 05)
- `Thread.MAX_PRIORITY` (giá trị 10)

Để lấy độ ưu tiên của 1 luồng, ta dùng phương thức: `int getPriority()`.

***Ví dụ 3-3**. Chương trình xác định mức độ ưu tiên của luồng.*

WorkingThread.java

```
package vn.tbit.info;

public class WorkingThread extends Thread {
    public WorkingThread(String name) {
        super(name);
    }

    public void run() {
        for (int i = 0; i &lt; 5; i++) {
            System.out.printf("Luồng: %s có độ ưu tiên là %d \n",
getName(), getPriority());
        }
    }
}
```

ThreadInfoExample.java

```
package vn.tbit.info;

public class ThreadInfoExample {

    public static void main(String[] args) {
```

```
        Thread t1 = new WorkingThread("Luồng 1");
        Thread t2 = new WorkingThread("Luồng 2");
        Thread t3 = new WorkingThread("Luồng 3");

        System.out.println("ID luồng 1: " + t1.getId());
        System.out.println("ID luồng 2: " + t2.getId());
        System.out.println("ID luồng 3: " + t3.getId());

        t1.setPriority(1);
        t2.setPriority(5);
        t3.setPriority(10);

        t1.start();
        t2.start();
        t3.start();
    }
}
```

Kết quả thực thi chương trình trên:

```
ID luồng 1: 10
ID luồng 2: 11
ID luồng 3: 12
Luồng: Luồng 2 có độ ưu tiên là 5
Luồng: Luồng 2 có độ ưu tiên là 5
Luồng: Luồng 2 có độ ưu tiên là 5
Luồng: Luồng 2 có độ ưu tiên là 5
Luồng: Luồng 2 có độ ưu tiên là 5
Luồng: Luồng 1 có độ ưu tiên là 1
Luồng: Luồng 3 có độ ưu tiên là 10
Luồng: Luồng 3 có độ ưu tiên là 10
Luồng: Luồng 3 có độ ưu tiên là 10
Luồng: Luồng 3 có độ ưu tiên là 10
Luồng: Luồng 3 có độ ưu tiên là 10
Luồng: Luồng 1 có độ ưu tiên là 1
Luồng: Luồng 1 có độ ưu tiên là 1
Luồng: Luồng 1 có độ ưu tiên là 1
Luồng: Luồng 1 có độ ưu tiên là 1
```

3.6.2 Sử dụng phương thức sleep()

Phương thức `sleep()` của lớp `Thread` được sử dụng để tạm ngừng một luồng trong một khoảng thời gian nhất định.

***Ví dụ 3-4**. Viết chương trình in ra số từ 1 đến 5, tạm ngừng 500ms trước khi in chữ số tiếp theo.*

```
package vn.tbit.sleep;

public class SleepMethodExample extends Thread {

    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(i);
            try {
```

```
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String args[]) {
        SleepMethodExample t1 = new SleepMethodExample();
        t1.start();
    }
}
```

3.6.3 Sử dụng join() và join(long millis)

Phương thức `join()` dùng để thông báo rằng hãy chờ luồng này hoàn thành rồi luồng cha mới được tiếp tục chạy.

***Ví dụ 3-5**. Chương trình minh họa sửa dụng phương thức `join()` của luồng.*

```
package vn.tbit.join;

public class UsingJoinMethod extends Thread {

    public UsingJoinMethod(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println(getName());
        for (int i = 1; i <= 5; i++) {
            try {
                System.out.print(i + " ");
                Thread.sleep(300);
            } catch (InterruptedException ie) {
                System.out.println(ie.toString());
            }
        }
        System.out.println();
    }

    public static void main(String[] args) throws InterruptedException {
        UsingJoinMethod t1 = new UsingJoinMethod("Thread 1");
        UsingJoinMethod t2 = new UsingJoinMethod("Thread 2");
        t1.start();
        t1.join();
        t2.start();
        System.out.println("Main Thread Finished");
    }
}
```

Thực thi chương trình trên:

```
Thread 1
```

```
1 2 3 4 5
Main Thread Finished
Thread 2
1 2 3 4 5
```

Phương thức `join(long millis)` dùng để thông báo luồng cha cần phải đợi sau `millis` (tính bằng *mili giây*) mới được tiếp tục chạy, kể từ lúc gọi `join(long millis)`. Nếu tham số bằng `0` nghĩa là đợi cho tới khi luồng này kết thúc.

***Ví dụ 3-6**. Chương trình minh họa sử dụng phương thức `join(long millis)`.*

```
package vn.tbit.join;

public class UsingJoinMethod2 extends Thread {

    public UsingJoinMethod2(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println(getName());
        for (int i = 1; i <= 5; i++) {
            try {
                System.out.print(i + " ");
                Thread.sleep(300);
            } catch (InterruptedException ie) {
                System.out.println(ie.toString());
            }
        }
        System.out.println();
    }

    public static void main(String[] args) throws InterruptedException {
        UsingJoinMethod2 t1 = new UsingJoinMethod2("Thread 1");
        UsingJoinMethod2 t2 = new UsingJoinMethod2("Thread 2");
        t1.start();

        // Main Thread phải chờ 450ms mới được tiếp tục chạy.
        // Không nhất thiết phải chờ Thread t1 kết thúc
        t1.join(450);
        t2.start();
        System.out.println("Main Thread Finished");
    }
}
```

Thực thi chương trình trên:

```
Thread 1
1 2 Main Thread Finished
Thread 2
1 3 2 4 3 5 4
5
```

3.6.4 Xử lý ngoại lệ cho luồng

Phương thức `Thread.setDefaultUncaughtExceptionHandler()` thiết lập mặc định xử lý khi luồng đột ngột chấm dứt do một ngoại lệ xảy ra mà không có xử lý khác đã được xác định cho luồng đó.

***Ví dụ 3-7**. Chương trình chưa xử lý ngoại lệ.*

```
package vn.tbit.exception;

import java.util.Random;

public class WorkingThread implements Runnable {

    @Override
    public void run() {
        while (true) {
            processSomething();
        }
    }

    private void processSomething() {
        try {
            System.out.println("Processing working thread");
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        Random r = new Random();
        int i = r.nextInt(100);
        if (i > 70) {
            throw new RuntimeException("Simulate an exception was not
handled in the thread");
        }
    }
}
```

***Ví dụ 3-8**. Chương trình minh họa xử lý ngoại lệ trong luồng.*

```
package vn.tbit.exception;

public class ThreadExceptionDemo {

    public static void main(String[] args) {
        System.out.println("==> Main thread running...");

        Thread thread = new Thread(new WorkingThread());
        Thread.setDefaultUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {
            @Override
            public void uncaughtException(Thread t, Throwable e) {
                System.out.println("#Thread: " + t);
```

```
                System.out.println("#Thread exception message: " +
e.getMessage());
            }
        });

        thread.start();
        System.out.println("==> Main thread end!!!");
    }
}
```

Thực thi chương trình trên:

```
==> Main thread running...
==> Main thread end!!!
Processing working thread
Processing working thread
Processing working thread
Processing working thread
Processing working thread
Processing working thread
#Thread: Thread[Thread-0,5,main]
#Thread exception message: Have a problem...
```

CÂU HỎI, BÀI TẬP VẬN DỤNG:

1. Luồng (thread) là gì? Lập trình đa luồng (multithread) có đặc điểm gì?
2. Đa nhiệm là gì? Các cách để đạt được đa nhiệm và phân biệt các cách đó?
3. Nêu ưu điểm và nhược điểm của lập trình đa luồng?
4. Trình bày các cách tạo luồng trong lập trình Java?
5. Viết một chương trình đa luồng dùng lớp *Thread*?
6. Viết một chương trình đa luồng dùng giao diện *Runnable*?

CHƯƠNG 4. LỚP INETADDRESS

Các thiết bị kết nối tới Internet được gọi là các *nút mạng* (node). Nếu nút là máy tính chúng ta gọi là *host*. Mỗi nút hoặc host được phân biệt với nhau bởi các địa chỉ mạng, chúng ta thường gọi là địa chỉ IP. Hầu hết địa chỉ IP hiện nay là địa chỉ IPv4 (địa chỉ IP phiên bản 4) có độ dài 4 byte. Mặc dù vậy, nhiều tổ chức và cá nhân đang dần chuyển sang sử dụng địa chỉ IPv6 (địa chỉ IP phiên bản 6) có độ dài 16 byte. Cả địa chỉ IPv4 và IPv6 đều bao gồm các byte được sắp thứ tự nhất định (có thể coi đó là một mảng các byte) nhưng chúng thực tế không phải là số.

Một địa chỉ IPv4 gồm 4 byte, mỗi byte thường được kí hiệu bằng một số nguyên dương có giá trị nằm trong khoảng từ 0 tới 255. Các byte được ngăn các bởi các dấu chấm để cho chúng ta dễ nhận ra chúng. Ví dụ, địa chỉ IP của trang *utb.edu.vn* là 117.6.86.168. Cách kí hiệu này thuật ngữ tiếng Anh gọi là định dạng *dotted quad*.

Một địa chỉ IPv6 thường được kí hiệu bởi 8 nhóm cách nhau bởi dấu hai chấm, trong đó mỗi nhóm gồm 4 số thập lục phân. Ví dụ, địa chỉ IP của trang *www.hamiltonweather.tk* là 2400:cb00:2048:0001:0000:0000:6ca2:c665. Những số 0 ở đầu mỗi nhóm có thể bỏ đi, do đó địa chỉ trên đây có thể viết là 2400:cb00:2048:1:0:0:6ca2:c665. Nếu trong địa chỉ IPv6 có một dãy các nhóm gồm toàn con số 0, chúng ta có thể dùng hai dấu hai chấm để kí hiệu thay thế. Ví dụ, địa chỉ 2001:4860:4860:0000:0000:0000:0000:8888 có thể được viết ngắn gọn là 2001:4860:4860::8888. Một cách viết địa chỉ IPv6 khác là kết hợp với cách viết địa chỉ IPv4 bằng cách viết 4 byte cuối của địa chỉ IPv6 dưới dạng địa chỉ IPv4. Ví dụ, địa chỉ FEDC:BA98:7654:3210:FEDC:BA98:7654:3210 có thể được viết là FEDC:BA98:7654:3210:FEDC:BA98:118.84.50.16.

Địa chỉ IP rất tiện lợi cho máy tính nhưng gây khó khăn cho con người trong việc ghi nhớ chúng. Vào những năm 50 của thế kỉ XX, G. A. Miller đã khám phá ra hầu hết mọi người có thể ghi nhớ một số có bảy chữ số; một vài người có thể nhớ nhiều hơn chín chữ số và một số người không thể nhớ quá năm chữ số. Chi tiết bạn đọc có thể tìm hiểu bài viết "*The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*" trong cuốn *Psychological Review*, tập 63, trang 81-97. Điều đó giải thích tại sao số điện thoại chúng ta thường chia thành các nhóm ba hoặc bốn số và bổ sung thêm mã vùng để giảm bớt các con số cần phải ghi nhớ trong mỗi nhóm. Do đó, một địa chỉ IPv4 có thể có tới mười hai chữ số là quá khó khăn cho hầu hết chúng ta để ghi nhớ chúng.

Để giúp chúng ta dễ dàng hơn trong việc ghi nhớ các địa chỉ mạng, các nhà thiết kế Internet đã phát minh ra các hệ thống DNS - Domain Name System (hệ thống máy chủ tên miền). DNS giúp chúng ta thay thế các kí hiệu theo dãy số bằng

những chuỗi kí tự dễ nhớ hơn với con người, chúng ta gọi chúng là *hostname*. Ví dụ, thay vì phải nhớ địa chỉ 117.6.86.168 chúng ta có thể nhớ địa chỉ *utb.edu.vn*. Mỗi một máy chủ phải có ít nhất một hostname. Máy khách cũng thường có một hostname nhưng không có một địa chỉ IP cố định nếu IP này được cấp phát lại sau mỗi lần khởi động.

Một số thiết bị có thể có nhiều tên. Ví dụ, cả *tbit.vn* và *tuhoctin.net* đều nằm trên một máy chủ Linux. Cái tên *tbit.vn* thực tế chỉ tới một website chứ không phải là tên một máy chủ cụ thể nào đó. Trong quá khứ, khi một website được chuyển từ một máy chủ tới một máy chủ khác thì tên của nó sẽ được gán lại trên máy chủ mới để luôn trỏ về website trên máy chủ hiện tại.

Có những trường hợp một tên có thể tương ứng với nhiều địa chỉ IP. Khi đó, máy chủ nào được lựa chọn để phản hồi yêu cầu từ người dùng sẽ được máy chủ DNS lựa chọn ngẫu nhiên. Tính năng này thường được sử dụng với những website có lượng truy cập rất lớn nên cần mở rộng hệ thống theo chiều ngang để phân chia lượng người dùng tới nhiều hệ thống máy chủ khác nhau. Ví dụ, tên miền *google.com.vn* thường trỏ tới nhiều máy chủ có địa chỉ khác nhau và thay đổi theo từng thời điểm.

Tất cả máy tính kết nối tới Internet đều truy cập một thiết bị gọi là *máy chủ dịch vụ tên miền* (máy chủ DNS). Đây là máy chủ thực hiện ánh xạ giữa tên miền và địa chỉ IP. Hầu hết máy chủ DNS chỉ biết những địa chỉ của những máy tính trong mạng cục bộ của mình và một số ít địa chỉ trên một mạng khác. Nếu máy khách yêu cầu địa chỉ của một máy nằm bên ngoài mạng cục bộ, máy chủ DNS của mạng cục bộ sẽ gửi yêu cầu tới một máy chủ DNS tại mạng khác đợi trả lời để phản hồi lại yêu cầu đó.

Hầu như mọi lúc chúng ta có thể sử dụng hostname và đợi máy chủ DNS xử lý trả lại địa chỉ IP. Khi nào kết nối tới máy chủ DNS, chúng ta không cần lo lắng về việc làm thế nào để có thể ánh xạ giữa tên và địa chỉ IP trên máy chủ DNS cục bộ hay các bộ phận khác trên Internet. Mặc dù vậy, chúng ta cần truy cập ít nhất một máy chủ DNS nếu muốn thực hành một số ví dụ trong giáo trình này. Một số ví dụ cần phải kết nối Internet chứ không làm việc trên một máy độc lập được.

Lớp `java.net.InetAddress` là một sự biểu diễn bậc cao của địa chỉ IP, bao gồm cả IPv4 và IPv6. Lớp này được dùng trong hầu hết các lớp khác như `Socket`, `ServerSocket`, `URL`, `DatagramSocket`, `DatagramPacket`... `InetAddress` cũng bao gồm các thông tin cả về hostname và địa chỉ IP.

4.1 Khởi tạo đối tượng InetAddress

Lớp `InetAddress` không có một hàm tạo `public` nào. Thay vào đó, `InetAddress` có một số phương thức `static` để kết nối tới máy chủ DNS. Phương thức thường được dùng nhất là `InetAddress.getByName()`. Ví dụ, đây là cách để chúng ta tạo một object để lấy thông tin của *utb.edu.vn*:

```
InetAddress address = InetAddress.getByName("utb.edu.vn");
```

Ví dụ 4-1. *Tạo một object của lớp* `InetAddress` *để đọc thông tin của tên miền* `utb.edu.vn`.

```
import java.net.*;
public class InetGetByName {
     public static void main (String[] args) {
          try {
               InetAddress address =
                              InetAddress.getByName("utb.edu.vn");
               System.out.println(address);
          } catch (UnknownHostException ex) {
               System.out.println("Khong tim thay!!!");
          }
     }
}
```

Kết quả:

```
% java InetGetByName
utb.edu.vn/117.6.86.168
```

Chúng ta cũng có thể sử dụng tham số là một địa chỉ IP. Ví dụ, lấy hostname tương ứng với địa chỉ IP 117.6.86.168:

```
InetAddress address = InetAddress.getByName("117.6.86.168");
System.out.println(address.getHostName());
```

Nếu địa chỉ IP chúng ta tìm không có hostname tương ứng, `getHostName()` sẽ đơn giản trả về địa chỉ IP dạng *dotted squad*.

Như bên trên đã nói, một số tên miền có thể tương ứng với nhiều địa chỉ IP. Phương thức để lấy thông tin về tất cả các máy chủ tương ứng là `getAllByName()`. Phương thức này sẽ trả về một mảng thông tin tương ứng với các máy chủ.

```
try {
     InetAddress[] addresses =
     InetAddress.getAllByName("google.com.vn");
     for (InetAddress address : addresses) {
          System.out.println(address);
     }
} catch (UnknownHostException ex) {
     System.out.println("Không tìm thấy thông tin");
}
```

Cuối cùng, phương thức `getLocalHost()` trả về thông tin của máy cục bộ (máy mà đoạn mã Java được thực thi).

```
 InetAddress me = InetAddress.getLocalHost();
```

Phương thức này sẽ cố gắng kết nối tới một máy chủ DNS để lấy thông tin về hostname và địa chỉ IP. Trong trường hợp không thành công nó sẽ trả về địa chỉ loopback. Địa chỉ IP tương ứng với *localhost* là 127.0.0.1.

Ví dụ 4-2*. Viết chương trình hiển thị thông tin của máy cục bộ.*

```
import java.net.*;
public class MyAddress {
     public static void main (String[] args) {
          try {
               InetAddress address = InetAddress.getLocalHost();
               System.out.println(address);
          } catch (UnknownHostException ex) {
               System.out.println("Không tìm thấy!!!");
          }
     }
}
```

Đây là kết quả trên máy tính của tôi. Trên máy của bạn sẽ là một kết quả khác.

```
% java MyAddress
Nguyens-MacBook-Pro.local/10.211.55.2
```

Như thấy ở kết quả trên, chúng ta có thể thấy hostname/IP của máy tính chạy chương trình. Nếu không kết nối tới Internet và không đặt địa chỉ IP tĩnh cho máy tính của mình, chúng ta sẽ nhận được kết quả hostname là *localhost* và địa chỉ IP là *127.0.0.1*.

Nếu biết chính xác địa chỉ IP của máy tính, chúng ta có thể tạo một `InetAddress` sử dụng phương thức `InetAddress.getByAddress()`. Phương thức này có thể chứa một hoặc 2 tham số.

```
public static InetAddress getByAddress(byte[] addr) throws
UnknownHostException
public static InetAddress getByAddress(String hostname, byte[] addr)
throws UnknownHostException
```

Phương thức `InetAddress.getByAddress()` thứ nhất tạo một `InetAddress` với một địa chỉ IP cho trước mà không cần tham số là hostname trong khi đó phương thức thứ hai thì cần tham số này. Ví dụ sau sẽ tạo một `InetAddress` cho địa chỉ 117.6.86.168:

```
byte[] address = {117, 6, 86, (byte) 168};
InetAddress tbit = InetAddress.getByAddress(address);
InetAddress tbitWithName = InetAddress.getByAddress("tbit.vn", address);
```

Chú ý rằng giá trị lớn hơn 127 phải chuyển thành kiểu `byte`.

Không giống với các phương thức khác, hai phương thức này không đảm bảo máy chủ đó tồn tại hoặc máy chủ đó được ánh xạ tới địa chỉ IP. Việc bẫy lỗi sẽ `throws` lớp `UnknownHostException` nếu kích thước byte không phù hợp (không phải là 4 byte hoặc 16 byte). Phương thức này sẽ hữu dụng nếu thông tin tên của Server không có sẵn hoặc không đúng. Ví dụ, khi không thể nhớ được tên của một máy trạm, một máy in mạng hay một router trong hệ thống mạng nội bộ của mình có địa chỉ IP là bao nhiêu, chúng ta có thể viết một chương trình nhỏ để kiểm tra các máy đang bật trong hệ thống mạng của mình và chỉ phải quét 254 trường hợp để tìm ra thiết bị đó.

4.2 Nhớ đệm (caching)

Bởi vì chi phí thời gian cho việc tìm kiếm DNS là khá đắt đỏ (mất vài giây để yêu cầu được gửi qua một loạt các máy chủ hoặc là có host không tìm thấy được), do đó lớp `InetAddress` lưu kết quả tìm kiếm vào vùng nhớ đệm. Khi nó có được địa chỉ của một host nào đó nó sẽ không thực hiện tìm kiếm lại thậm chí cả khi chúng ta tạo ra một `InetAddress` mới cho cùng một host. Địa chỉ IP của host đó sẽ không thay đổi khi chương trình đang chạy.

Với những kết quả không tốt (như không tìm thấy host), lớp `InetAddress` sẽ chỉ lưu giữ kết quả này trong 10 giây.

Thời gian lưu giữ các kết quả tạm thời có thể được điều chỉnh trong lớp `networkaddress.cache.ttl` và lớp `networkaddress.cache.negative.ttl`. Lớp `networkaddress.cache.ttl` sẽ quy định thời gian tồn tại của những truy vấn DNS thành công trong khi lớp còn lại sẽ quy định thời gian tồn tại của những truy vấn DNS không thành công. Nếu cố gắng tìm kiếm cùng một host trong khoảng thời gian quy định sẽ cho về cùng một kết quả. Giá trị `-1` sẽ đồng nghĩa với việc "*không bao giờ hết hạn*".

Bên cạnh việc lưu kết quả trong vùng nhớ đệm của lớp `InetAddress`, máy cục bộ, máy chủ tên miền cục bộ hoặc các máy chủ DNS cũng có thể lưu kết quả của mình trong một vùng nhớ đệm của nó trong một khoảng thời gian nhất định. Java không thể can thiệp vào việc lưu giữ kết quả tạm thời này. Do đó, có thể mất nhiều giờ để thay đổi địa chỉ IP tương ứng với *host* (hoặc tên miền) trên Internet là điều rất có thể xảy ra. Trong khoảng thời gian ấy, chương trình của chúng ta cũng có thể gặp phải những lỗi xảy ra, bao gồm các ngoại lệ như `UnknownHostException`, `NoRouteToHostException` và `ConnectException`.

4.3 Tìm kiếm bằng địa chỉ IP

Khi chúng ta gọi phương thức `getByName()` với tham số là một địa chỉ IP, nó sẽ tạo ra một đối tượng `InetAddress` tương ứng với địa chỉ IP mà không kiểm tra DNS. Điều đó có nghĩa là nó có thể tạo một đối tượng `InetAddress` cho một host không thực sự tồn tại và chúng ta không thể kết nối tới nó. *Hostname* của đối tượng `InetAddress` đã tạo ra từ một xâu chứa địa chỉ IP được khởi tạo từ chính xâu đó. Một truy vấn DNS chỉ thực sự diễn ra khi có yêu cầu hostname từ đối tượng đó hoặc là qua phương thức `getHostName()`. Như vậy thì *tbit.vn* được xác định từ địa chỉ IP 45.117.83.115 như thế nào. Nếu tại thời điểm hostname được yêu cầu và một tìm kiếm DNS được thực hiện, host tương ứng với địa chỉ IP không thể được tìm thấy thì hostname được gán là chuỗi IP được khai báo. Bẫy lỗi `UnknownHostException` không được thực hiện.

Hostname có tính ổn định hơn nhiều so với địa chỉ IP. Một số dịch vụ có thể chạy trên một hostname nhiều năm nhưng đổi địa chỉ IP nhiều lần. Nêu được lựa chọn giữa hostname và IP thì chúng ta nên chọn hostname, chỉ lựa chọn IP khi hostname không có sẵn.

4.4 Các phương thức Get

Lớp `InetAddress` chứa bốn phương thức Get để trả lại chuỗi hostname, địa chỉ IP cả dưới dạng chuỗi và dạng mảng byte.

```
public String getHostName()
public String getCanonicalHostName()
public byte[] getAddress()
public String getHostAddress()
```

Không có các phương thức `setHostName()` và `setAddress()` tương ứng, có nghĩa là chúng ta không thể thay đổi các trường đó từ bên ngoài gói `java.net`. Điều đó giúp cho `InetAddress` được bảo vệ tốt hơn.

Phương thức `getHostName()` trả về một xâu chứa tên của host với địa chỉ IP được đại diện cho đối tượng `InetAddress`. Nếu thiết bị tương ứng không có hostname hoặc bị ngăn chặn vì vấn đề bảo mật thì phương thức này sẽ trả về xâu địa chỉ IP. Ví dụ:

```
InetAddress machine = InetAddress.getLocalHost();
String localhost = machine.getHostName();
```

Phương thức `getCanonicalHostName()` tương tự phương thức trên nhưng nó mạnh hơn một chút trong việc truy vấn DNS. Phương thức `getHostName()` chỉ thực hiện gọi DNS khi nó không chắc chắn về hostname. Trong khi đó phương thức

getCanonicalHostName() gọi DNS khi nó có thể và có thể sẽ thay thế các kết quả trước đó về việc tìm hostname. Ví dụ:

```
InetAddress machine = InetAddress.getLocalHost();
String localhost = machine.getCanonicalHostName();
```

Phương thức *getCanonicalHostName()* hữu ích khi bắt đầu với một địa chỉ IP thay vì hostname. Trong ví dụ ở dưới, địa chỉ IP 45.117.83.115 được dùng để tạo một InetAddress thông qua phương thức *getByName()* và sau đó phương thức *getCanonicalHostName()* sẽ được dùng để lấy được hostname.

***Ví dụ 4-3**. Cho địa chỉ, tìm tên miền (tên host).*

```
import java.net.*;
public class ReverseTest {
     public static void main (String[] args) throws UnknownHostException
{
          InetAddress ia = InetAddress.getByName("98.138.219.231");
          System.out.println(ia.getCanonicalHostName());
     }
}
```

Kết quả sẽ trả về (tại thời điểm viết):

```
% java ReverseTest
media-router-fp1.prod1.media.vip.ne1.yahoo.com
```

Phương thức *getHostAddress()* sẽ trả về một xâu chứa địa chỉ IP tương ứng. Ví dụ bên dưới là một ví dụ về việc sử dụng phương thức này.

***Ví dụ 4-4**. Tìm địa chỉ IP của máy cục bộ*

```
import java.net.*;
public class MyAddress {
     public static void main(String[] args) {
          try {
               InetAddress me = InetAddress.getLocalHost();
               String dottedQuad = me.getHostAddress();
               System.out.println("Địa chỉ máy cục bộ: " + dottedQuad);
          } catch (UnknownHostException ex) {
               System.out.println("Không tìm thấy kết quả.");
          }
     }
}
```

Kết quả (khác nhau trên các máy khác nhau kết nối tới các mạng khác nhau):

```
% java MyAddress
Địa chỉ máy cục bộ: 192.168.1.21
```

Tất nhiên là địa chỉ IP ở kết quả trên phụ thuộc vào nơi mà đoạn mã được thực thi.

Nếu muốn biết địa chỉ IP của một máy (hiếm khi dùng) thì chúng ta sử dụng phương thức *getAddress()*. Phương thức này sẽ trả về một mảng kiểu byte biểu diễn địa chỉ IP của máy. Byte quan trọng nhất (hay byte đầu tiên trong địa chỉ IP) là byte đầu tiên. Nếu muốn biết chiều dài của mảng, hãy sử dụng thuộc tính *length* của mảng (sử dụng để kiểm tra loại địa chỉ IPv4 hoặc IPv6).

```
InetAddress me = InetAddress.getLocalHost();
byte[] address = me.getAddress();
```

Chúng ta biết là địa chỉ IP được biểu diễn số nguyên không dấu. Nhưng không giống ngôn ngữ C, kiểu *byte* của Java có phạm vi từ -128 đến 127. Có nghĩa là những số lớn hơn 127 là những số âm trong kiểu *byte* của Java. Do đó, nếu muốn sử dụng kết quả kiểu *byte* được trả về bởi phương thức *getAddress()* để làm gì đó, chúng ta cần chuyển kiểu *byte* sang kiểu *int* bằng cách điều chỉnh thích hợp. Ví dụ như sau:

```
int unsignedByte = signedByte < 0 ? signedByte + 256 : signedByte;
```

Trong ví dụ trên, *signedByte* có thể âm hoặc dương. Phép toán điều kiện *?* sẽ kiểm tra xem nó có âm hay không. Nếu âm, nó sẽ được cộng với 256 để trở thành số dương kiểu *byte* và ngược lại thì giữ nguyên.

Một lý do để sử dụng mảng *byte* trả về bởi phương thức *getAddress()* là để xác định loại địa chỉ IPv4 hay IPv6.

Ví dụ 4-5*. Xác định địa chỉ IP v4 hay v6.*

```
import java.net.*;
public class AddressTests {
     public static int getVersion(InetAddress ia) {
          byte[] address = ia.getAddress();
          if (address.length == 4) return 4;
          else if (address.length == 16) return 6;
          else return -1;
     }
}
```

4.5 Kiểm tra loại địa chỉ

Một số địa chỉ IP và một số dải địa chỉ IP có ý nghĩa đặc biệt. Ví dụ, địa chỉ 127.0.0.1 luôn là địa chỉ loopback. Địa chỉ loopback, còn gọi là localhost, là địa chỉ trỏ chính máy hiện tại. Địa chỉ IPv4 trong phạm vi 224.0.0.0 tới 239.255.255.255 là địa chỉ multicast dùng để gửi thông điệp tới nhiều host cùng lúc. Java có 10 phương thức dùng để xác định các loại địa chỉ đặc biệt.

- *public boolean* ***isAnyLocalAddress****()*
- *public boolean* ***isLoopbackAddress****()*

- `public boolean` ***`isLinkLocalAddress`***`()`
- `public boolean` ***`isSiteLocalAddress`***`()`
- `public boolean` ***`isMulticastAddress`***`()`
- `public boolean` ***`isMCGlobal`***`()`
- `public boolean` ***`isMCNodeLocal`***`()`
- `public boolean` ***`isMCLinkLocal`***`()`
- `public boolean` ***`isMCSiteLocal`***`()`
- `public boolean` ***`isMCOrgLocal`***`()`

Phương thức `isAnyLocalAddress()` trả về giá trị true nếu địa chỉ là địa chỉ *wildcard*, ngược lại trả về giá trị false. Địa chỉ *wildcard* khớp với bất kỳ địa chỉ nào trên hệ thống mạng cục bộ. Điều này sẽ quan trọng khi một hệ thống có nhiều giao diện mạng, như trường hợp hệ thống máy tính có nhiều card Ethernet và/hoặc card WiFi 802.11. Trong IPv4, địa chỉ wildcard là 0.0.0.0. Trong IPv6, địa chỉ wildcard là 0:0:0:0:0:0:0:0 (hoặc kí hiệu là ::).

Phương thức `isLinkLocalAddress()` trả về giá trị true nếu địa chỉ này là một địa chỉ IPv6 dạng link-local. Đây là một địa chỉ giúp mạng IPv6 tự cấu hình, giống như DHCP trong mạng IPv4 nhưng không nhất thiết phải dùng một Server trong mạng này. Tất cả địa chỉ dạng link-local đều bắt đầu bởi tám byte FE00:0000:0000:0000. Tám byte tiếp theo là địa chỉ cục bộ, thường được sao chép thừ địa chỉ Ethernet MAC được gán bởi nhà sản xuất thiết bị mạng.

Phương thức `isSiteLocalAddress()` trả về giá trị true nếu một địa chỉ IPv6 là địa chỉ site-local. Địa chỉ loại này bắt đầu bởi tám byte FEC0:0000:0000:0000. Tám byte tiếp theo cũng được sao chép từ địa chỉ Ethernet MAC.

Phương thức `isMulticastAddress()` trả về giá trị true nếu địa chỉ là multicast. Những máy có địa chỉ này sẽ gửi nội dung quảng bá tới một số nhất định các máy khác đã đăng kí thay vì chỉ gửi cho một máy nhất định. Trong IPv4, dải địa chỉ multicast từ 224.0.0.0 đến 239.255.255.255. Trong IPv6, chúng luôn được bắt đầu bởi FF.

Phương thức `isMCGlobal()` trả về giá trị true nếu địa chỉ này là địa chỉ multicast toàn cục (global). Địa chỉ multicast toàn cục sẽ quảng bá tới các máy đăng kí trên toàn bộ hệ thống mạng toàn cầu. Tất cả địa chỉ loại này đều bắt đầu bằng FE. Trong IPv6, chúng bắt đầu bởi FF0E hoặc FF1E phụ thuộc vào việc đó là địa chỉ multicast vĩnh viễn hoặc tạm thời. Trong IPv4, tất cả địa chỉ IPv4 đều có phạm vi toàn cầu.

Phương thức `isMCOrgLocal()` trả về giá trị true nếu nó là địa chỉ dạng multicast tổ chức. Địa chỉ multicast loại này là địa chỉ multicast mà tất cả các máy

đăng kí thuộc trong cùng một cơ quan, tổ chức chứ không có máy từ bên ngoài. Địa chỉ loại này bắt đầu bởi FF08 hoặc FF18 phụ thuộc vào địa chỉ multicast là vĩnh viễn hoặc tạm thời.

Phương thức `isMCSiteLocal()` trả về giá trị true nếu nó là địa chỉ dạng multicast site-wide. Những packet tới máy có địa chỉ loại này chỉ được truyền trong mạng cục bộ của nó. Địa chỉ loại này bắt đầu bởi FF05 hoặc FF15 phụ thuộc vào địa chỉ multicast là vĩnh viễn hoặc tạm thời.

Phương thức `isMCLinkLocal()` trả về giá trị true nếu nó là địa chỉ multicast subnet-wide. Packet tới địa chỉ loại này chỉ được truyền trong nội bộ mạng con (subnet). Địa chỉ loại này bắt đầu bởi FF02 hoặc FF12 phụ thuộc vào địa chỉ multicast là vĩnh viễn hoặc tạm thời.

Phương thức `isMCNodeLocal()` trả về giá trị true nếu nó là địa chỉ multicast dạng interface-local. Packet tới địa chỉ loại này không thể truyền ra khỏi giao diện mạng của nó, thậm chí không thể tới một giao diện mạng khác trong cùng node. Địa chỉ loại này bắt đầu bởi FF01 hoặc FF11 phụ thuộc vào địa chỉ multicast là vĩnh viễn hoặc tạm thời.

Ví dụ 4-6 sau đây là một chương trình đơn giản để kiểm tra địa chỉ nhập trực tiếp từ cửa sổ lệnh dùng 10 phương thức trên.

***Ví dụ 4-6**. Kiểm tra địa chỉ IP đặc trưng*

```
import java.net.*;

public class IPCharacteristics {

    public static void main(String[] args) {
        try {
            InetAddress address = InetAddress.getByName(args[0]);
            if (address.isAnyLocalAddress()) {
                System.out.println(address + " là địa chỉ wildcard.");
            }
            if (address.isLoopbackAddress()) {
                System.out.println(address + " là địa chỉ loopback.");
            }
            if (address.isLinkLocalAddress()) {
                System.out.println(address + " là địa chỉ link-local.");
            } else if (address.isSiteLocalAddress()) {
                System.out.println(address + " là địa chỉ site-local.");
            } else {
                System.out.println(address + " là địa chỉ toàn cục.");
            }
            if (address.isMulticastAddress()) {
                if (address.isMCGlobal()) {
                    System.out.println(address + " là địa chỉ multicast
toàn cục.");
```

```
                } else if (address.isMCOrgLocal()) {
                    System.out.println(address + " là địa chỉ multicast
tổ chức.");
                } else if (address.isMCSiteLocal()) {
                    System.out.println(address + " là địa chỉ multicast
site-wide.");
                } else if (address.isMCLinkLocal()) {
                    System.out.println(address + " là địa chỉ multicast
subnet-wide.");
                } else if (address.isMCNodeLocal()) {
                    System.out.println(address + " là địa chỉ multicast
interface-local.");
                } else {
                    System.out.println(address + " là địa chỉ multicast
chưa xác định.");
                }
            } else {
                System.out.println(address + " là địa chỉ unicast.");
            }
        } catch (UnknownHostException ex) {
            System.err.println("Không phân tích được địa chỉ này.");
        }
    }
}
```

Dưới đây là một số kết quả khi chạy đoạn mã trên với các địa chỉ IPv4 và IPv6 khác nhau.

```
$ java IPCharacteristics 127.0.0.1
/127.0.0.1 là địa chỉ loopback.
/127.0.0.1 là địa chỉ toàn cục.
/127.0.0.1 là địa chỉ unicast.
$ java IPCharacteristics 192.168.254.32
/192.168.254.32 là địa chỉ multicast site-wide.
/192.168.254.32 là địa chỉ unicast.
$ java IPCharacteristics www.oreilly.com
www.oreilly.com/208.201.239.37 là địa chỉ toàn cục.
www.oreilly.com/208.201.239.37 là địa chỉ unicast.
$ java IPCharacteristics 224.0.2.1
/224.0.2.1 là địa chỉ toàn cục.
/224.0.2.1 là địa chỉ multicast toàn cục.
$ java IPCharacteristics FF01:0:0:0:0:0:0:1
/ff01:0:0:0:0:0:0:1 là địa chỉ toàn cục.
/ff01:0:0:0:0:0:0:1 là địa chỉ multicast interface-local.
$ java IPCharacteristics FF05:0:0:0:0:0:0:101
/ff05:0:0:0:0:0:0:101 là địa chỉ toàn cục.
/ff05:0:0:0:0:0:0:101 là địa chỉ multicast site-wide.
$ java IPCharacteristics 0::1
/0:0:0:0:0:0:0:1 là địa chỉ loopback.
/0:0:0:0:0:0:0:1 là địa chỉ toàn cục.
/0:0:0:0:0:0:0:1 là địa chỉ unicast.
```

4.6 Kiểm tra khả năng kết nối (reachable)

Lớp *InetAddress* có hai phương thức *isReachable()* để kiểm tra xem một nút mạng cụ thể có kết nối từ host hiện tại được không. Kết nối có thể bị khóa vì rất nhiều lí do, có thể là firewall, máy chủ proxy, đứt cáp hay host không hoạt động tại thời điểm chúng ta thử kết nối.

```
public boolean isReachable(int timeout) throws IOException
public boolean isReachable(NetworkInterface interface, int ttl, int
timeout) throws IOException
```

Hai phương thức này sử dụng công cụ truy vết *traceroute* để chỉ ra một địa chỉ có phản hồi hay không. Nếu host được kiểm tra trả lời trong thời gian *timeout* tính bằng mili giây, phương thức sẽ trả về giá trị *true* và ngược lại trả về giá trị *false*. Ngoại lệ *IOException* sẽ được gọi nếu xảy ra một lỗi mạng. Phương thức thứ hai có thể tham số *NetworkInterface* để chỉ định giao diện mạng được sử dụng và thời gian tồn tại *ttl* (time-to-live) là giá trị số lớn nhất xác định thời gian kết nối trước khi nó bị bỏ qua.

4.7 Các phương thức của Object

Giống như những lớp khác trong Java, *java.net.InetAddress* kế thừa từ lớp *java.lang.Object*. Vì vậy, lớp này cũng kế thừa tất cả các phương thức của lớp này. Lớp *InetAddress* override ba phương thức sau:

- *public boolean **equals**(Object o)*
- *public int **hashCode**()*
- *public String **toString**()*

Hai object của lớp *InetAddress* bằng nhau nếu chúng có cùng địa chỉ IP. Ví dụ, một object *InetAddress* cho *tbit.vn* sẽ bằng với một object cho *tuhoctin.net* vì cả hai tên miền này đều được trỏ về địa chỉ một địa chỉ IP. *Ví dụ 4-7* sẽ tạo các object tương ứng với hai tên miền này và sẽ cho chúng ta biết chúng cùng một địa chỉ IP nếu không có gì thay đổi.

***Ví dụ 4-7**. Kiểm tra **tbit.vn** và **tuhoctin.net** có cùng địa chỉ IP không?*

```
import java.net.*;

public class IBiblioAliases {
    public static void main(String args[]) {
        try {
            InetAddress ibiblio = InetAddress.getByName("tbit.vn");
            InetAddress helios = InetAddress.getByName("tuhoctin.net");
            if (ibiblio.equals(helios)) {
                System.out.println("tbit.vn thuộc cùng máy chủ với
tuhoctin.net");
```

```
            } else {
                System.out.println("tbit.vn không thuộc cùng máy chủ với
tuhoctin.net");
            }
        } catch (UnknownHostException ex) {
            System.out.println("Không tìm thấy host tương ứng.");
        }
    }
}
```

Kết quả chạy đoạn code trên tại thời điểm viết tài liệu này:

```
% java IBiblioAliases
tbit.vn thuộc cùng máy chủ với tuhoctin.net
```

Phương thức *hashcode()* trả về một số nguyên tương ứng với địa chỉ IP. Nếu hai object *InetAddress* cùng địa chỉ IP, chúng sẽ cùng mã hash code, mặc dù hostname có thể khác nhau.

Phương thức *toString()* trả về một đoạn văn bản ngắn mô tả object. *Ví dụ 4-1* và *Ví dụ 4-2* đã thực hiện gọi phương thức này. Kết quả trả về của nó dạng:

```
hostname/địa chỉ IP
```

Không phải tất cả object *InetAddress* đều có hostname. Nếu chúng không có hostname, địa chỉ IP sẽ được dùng thay thế trong phiên bản Java 1.3 trở về trước, còn đối với Java 1.4 trở về sau nó sẽ trả về một xâu rỗng.

4.8 Inet4Address và Inet6Address

Java sử dụng hai lớp *Inet4Address* và *Inet6Address* để phân biệt địa chỉ IPv4 và địa chỉ IPv6.

- *public final class **Inet4Address** extends InetAddress*
- *public final class **Inet6Address** extends InetAddress*

Hầu hết mọi khi, chúng ta không thực sự nên quan tâm tới địa chỉ IPv4 hay IPv6. Trong tầng ứng dụng, chúng ta không cần phải biết về điều đó. Nhưng cũng có nhiều khi chúng ta cần chúng để làm mọi việc nhanh hơn. Lớp *Inet4Address* override nhiều phương thức của lớp *InetAddress* không thay đổi tới các phương thức *public*. Lớp *Inet6Address* cũng tương tự nhưng nó thêm một phương thức không có trong lớp cha:

*public boolean **isIPv4CompatibleAddress**()*

Phương thức này trả về giá trị đúng nếu và chỉ nếu đó là một địa chỉ IPv4 có thể nằm trong một IPv6, có nghĩa là chỉ có 4 byte cuối là khác không. Do đó, địa chỉ IP sẽ có dạng 0:0:0:0:0:0:0:xxxx. Trong trường hợp này chúng ta cũng có thể tách

lấy bốn byte cuối cùng từ phương thức *getBytes()* để tạo ra một object *Inet4Address*. Tuy nhiên chúng ta hiếm khi thực hiện điều này.

4.9 Lớp NetworkInterface

Lớp *NetwordInterface* đại diện cho một địa chỉ IP cục bộ. Nó có thể là một giao diện mạng vật lý như một card Ethernet bổ sung (firewall hoặc router) hay một giao diện mạng ảo của cùng một thiết bị phần cứng với một địa chỉ IP khác. Lớp *NetworkInterface* liệt kê tất cả các địa chỉ cục bộ và tạo ra các object *InetAddress* tương ứng. Những object đó sẽ được dùng để tạo socket cho Client hoặc Server.

Một số phương thức

Vì object *NetworkInterface* đại diện cho một phần cứng vật lý hoặc địa chỉ mạng ảo nên có không được khởi tạo một cách tùy tiện. Giống như lớp *InetAddress*, có một số phương thức dùng để tạo ra object *NetworkInterface* với từng giao diện mạng cụ thể. Chúng ta có thể tạo object *NetworkInterface* từ một địa chỉ IP, từ hostname hoặc enumeration.

public static NetworkInterface ***getByName****(String name) throws SocketException*

Phương thức *getByName()* trả về một object *NetworkInterface* từ một tên cụ thể. Nếu không có một giao diện mạng nào tương ứng, nó sẽ trả về giá trị *null*. Rất hiếm khi xảy ra lỗi, nhưng nếu có chúng sẽ được xử lý bởi *SocketException*.

Định dạng tên phụ thuộc vào hệ điều hành. Trên các dòng hệ điều hành Unix, tên của các giao diện Ethernet có dạng eth0, eth1,… Địa chỉ *loopback* có tên dạng "lo". Trên dòng hệ điều hành Windows, tên là chuỗi dạng "CE31" và "ELX100" phụ thuộc vào tên của Nhà sản xuất và dòng sản phẩm phần cứng gắn với giao diện mạng. Dưới đây là ví dụ về thao tác với *NetworkInterface* với dòng hệ điều hành Unix:

```
try {
    NetworkInterface ni = NetworkInterface.getByName("eth0");
    if (ni == null) {
        System.err.println("No such interface: eth0");
    }
} catch (SocketException ex) {
    System.err.println("Could not list sockets.");
}
```

public static NetworkInterface ***getByInetAddress****(InetAddress address) throws SocketException*

Phương thức *getByInetAddress()* trả về một object *NetworkInterface* tương ứng với một địa chỉ IP cụ thể. Nếu không có giao diện mạng nào tương ứng tại máy cục bộ, nó sẽ trả về giáo trị *null*. Nếu có lỗi nào đó, nó sẽ throws *SocketException*. Dưới đây là một ví dụ tìm một giao diện mạng tương ứng với địa chỉ loopback:

```
try {
    InetAddress local = InetAddress.getByName("127.0.0.1");
    NetworkInterface ni = NetworkInterface.getByInetAddress(local);
    if (ni == null) {
        System.err.println("That's weird. No local loopback address.");
    }
} catch (SocketException ex) {
    System.err.println("Could not list network interfaces.");
} catch (UnknownHostException ex) {
    System.err.println("That's weird. Could not lookup 127.0.0.1.");
}
```

public static Enumeration ***getNetworkInterfaces****() throws SocketException*

Phương thức *getNetworkInterfaces()* trả về một đối tượng thuộc lớp *java.util.Enumeration* liệt kê tất cả các giao diện mạng trên máy cục bộ. *Ví dụ 4-8* là một chương trình đơn giản để liệt kê các giao diện mạng trên máy cục bộ.

Ví dụ 4-8*. Liệt kê các giao diện mạng của máy cục bộ.*

```
import java.net.*; import java.util.*;
public class InterfaceLister {
     public static void main(String[] args) throws SocketException {
        Enumeration<NetworkInterface> interfaces =
NetworkInterface.getNetworkInterfaces();
        while (interfaces.hasMoreElements()) {
           NetworkInterface ni = interfaces.nextElement();
           System.out.println(ni);
        }
    }
}
```

Và kết quả với máy có nhiều kết nối mạng sẽ như sau:

```
% java InterfaceLister
name:eth1 (eth1) index: 3 addresses:
/192.168.210.122;
name:eth0 (eth0) index: 2 addresses:
/152.2.210.122;
name:lo (lo) index: 1 addresses:
/127.0.0.1;
```

Trong kết quả trên chúng ta có thể thấy máy tính có 2 kết nối mạng thông qua Ethernet với địa chỉ IP lần lượt là 192.168.210.122 và 152.2.210.122. Địa chỉ loopback cho localhost luôn là 127.0.0.1.

CÂU HỎI, BÀI TẬP VẬN DỤNG:

1. Lớp *InetAddress* trong Java dùng để làm gì? Các tạo đối tượng thuộc lớp *InetAddress*.
2. Liệt kê các phương thức quan trọng của lớp *InetAddress*?
3. Liệt kê các loại địa chỉ IP và cách kiểm tra chúng với Java?
4. Lớp *NetworkInterface* dùng để làm gì? Nêu các phương thức chính của nó?
5. Viết chương trình đọc thông tin của máy cục bộ: hostname, IP…?
6. Viết chương trình tìm địa chỉ IP của một tên miền hoặc tên máy trong mạng?
7. Viết chương trình liệt kê tất cả IP tương ứng với một tên miền?

CHƯƠNG 5. LẬP TRÌNH VỚI GIAO THỨC TCP

5.1 Khái niệm chung

Thuật ngữ lập trình mạng với Java đề cập đến việc viết các chương trình thực hiện trên nhiều thiết bị máy tính, trong đó các thiết bị được kết nối với nhau.

Gói `java.net` của Java chứa một tập hợp các lớp và giao tiếp cung cấp giao thức truyền thông ở mức độ thấp.

Gói `java.net` được cung cấp hỗ trợ cho hai giao thức mạng phổ biến sau:

- **TCP** - Transmission Control Protocol: TCP thường được sử dụng qua giao thức Internet (Internet Protocol), được gọi là TCP/IP. Giao thức này cho phép giao tiếp tin cậy giữa hai ứng dụng.
- **UDP** - User Datagram Protocol: một giao thức khác cho phép truyền dữ liệu giữa các ứng dụng. Giao thức này không kiểm tra đến việc gói tin đã được gửi hay chưa nên đây là giao tiếp không tin cậy giữa hai hoặc nhiều ứng dụng. Chúng ta sẽ tìm hiểu về lập trình với giao thức UDP ở chương sau.

TCP và UDP là các giao thức cốt lõi của việc kết nối các thiết bị công nghệ với nhau. Các ứng dụng có thể dùng một trong hai hoặc cả hai giao thức này để trao đổi với các ứng dụng trên máy tính khác thông qua mạng máy tính.

5.2 Khái niệm cổng (port number)

Để có thể thực hiện các cuộc giao tiếp, một trong hai quá trình phải công bố số hiệu cổng của socket mà mình sử dụng. Mỗi cổng giao tiếp thể hiện một địa chỉ xác định trong hệ thống. Khi quá trình được gán một số hiệu cổng, nó có thể nhận dữ liệu gửi đến cổng này từ các quá trình khác. Quá trình còn lại cũng được yêu cầu tạo ra một socket.

Số hiệu cổng (port number) được sử dụng để xác định tính duy nhất của các ứng dụng khác nhau. Nó hoạt động như một điểm kết nối cuối trong giao tiếp giữa các ứng dụng.

Số hiệu cổng gán cho Socket phải duy nhất trên phạm vi máy tính đó, có giá trị trong khoảng từ 0 đến 65535 (16 bit). Trong đó, giá trị cổng:

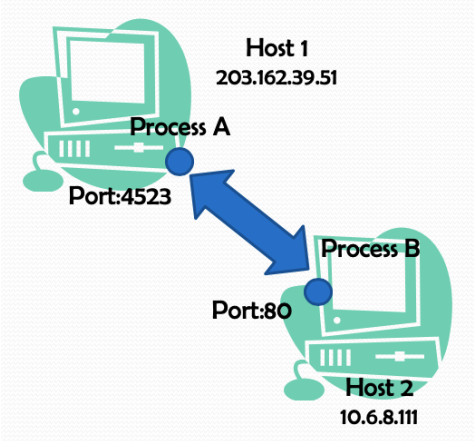
- Từ 0-1023: là cổng hệ thống (common hay well-known port), được dành riêng cho các quá trình của hệ thống.

- Từ 1024-49151: là cổng phải đăng ký (registered port). Các ứng dụng muốn sử dụng cổng này phải đăng ký với IANA (Internet Assigned Numbers Authority).
- Từ 49152-65535: là cổng dùng riêng hay cổng động (dynamic hay private port). Người sử dụng có thể dùng cho các ứng dụng của mình, không cần phải đăng ký.

Một số cổng thường được sử dụng:

- 21: dịch vụ FTP
- 23: dịch vụ Telnet
- 25: dịch vụ Email (SMTP)
- 80: dịch vụ Web (HTTP)
- 110: dịch vụ Email (POP)
- 143: dịch vụ Email (IMAP)
- 443: dịch vụ SSL (HTTPS)
- 1433/1434: cơ sở dữ liệu SQL Server
- 3306: cơ sở dữ liệu MySQL

5.3 Lớp Socket



Đơn vị truyền và nhận tin bằng phương thức TCP được gọi là Socket.

Socket cho phép dữ liệu được trao đổi giữa các thiết bị trong môi trường mạng máy tính.

Lớp `java.net.Socket` trong Java giúp quản lý quá trình truyền và nhận giữa các máy tính trong mạng máy tính bằng giao thức TCP.

Một Socket đóng vai trò một đầu-cuối của một kết nối thực. Một Socket vừa có thể của Client để gửi yêu cầu kết nối tới Server vừa có thể được tạo bởi Server để xử lý yêu cầu trao đổi tin từ Client.

Chúng ta cùng tìm hiểu các phương thức của lớp `Socket`.

5.3.1 Các phương thức tạo

```
public Socket(String host, int port) throws IOException,
UnknownHostException
```

Constructor này cố gắng để kết nối với máy chủ được chỉ định tại cổng được chỉ định. Nếu constructor này không ném một ngoại lệ, kết nối thành công và máy khách được kết nối với máy chủ.

```
public Socket(InetAddress host, int port)throws IOException,
UnknownHostException
```

Constructor này giống hệt với hàm tạo trước đó, ngoại trừ việc máy chủ được chỉ định bởi một đối tượng `InetAddress`.

```
public Socket(String host, int port, InetAddress
localAddress, int localPort)throws IOException
```

Kết nối đến máy chủ và cổng được chỉ định, tạo một socket trên máy chủ cục bộ tại địa chỉ và cổng được chỉ định.

```
public Socket(InetAddress host, int port, InetAddress
localAddress, int localPort) throws IOException
```

Constructor này giống hệt với constructor trước đó, ngoại trừ máy chủ được chỉ định bởi một đối tượng `InetAddress` thay vì một `String`.

```
public Socket()
```

Tạo một socket không chỉ định trước kết nối. Sau này chúng ta sử dụng phương thức `connect()` để kết nối socket này với máy chủ.

5.3.2 Các phương thức kiểm soát vào-ra

```
public InputStream getInputStream() throws IOException
```

Trả về dòng đầu vào của socket. Input stream được kết nối với output stream của socket remote.

```
public OutputStream getOutputStream() throws IOException
```

Trả về dòng đầu ra của socket. Output stream được kết nối với input stream của socket remote.

5.3.3 Một số phương thức khác

```
public void connect(SocketAddress host, int timeout) throws
IOException
```

Phương thức này kết nối socket với máy chủ được chỉ định. Phương thức này là cần thiết chỉ khi chúng ta khởi tạo Socket bằng cách sử dụng constructor không có đối số.

`public InetAddress` ***`getInetAddress`***`()`

Phương thức này trả về địa chỉ mạng của máy chủ mà socket này được kết nối.

`public int` ***`getPort`***`()`

Trả về cổng mà socket bị ràng buộc trên máy remote.

`public int` ***`getLocalPort`***`()`

Trả về cổng mà socket bị ràng buộc trên máy local.

`public SocketAddress` ***`getRemoteSocketAddress`***`()`

Trả về địa chỉ của socket từ xa.

`public synchronized void` ***`setSoTimeout`***`(int timeout) throws SocketException`

Thiết lập thời gian tồn tại của socket. Nếu `timeout` khác `0`, đây chính là khoảng thời gian (được tính bằng mili giây) mà socket còn hoạt động. Hết thời gian này chương trình socket sẽ tự hủy.

`public void` ***`close`***`() throws IOException`

Đóng socket, làm cho đối tượng `Socket` này không còn có khả năng kết nối với bất kỳ máy chủ nào.

5.4 Lớp ServerSocket

Lớp `java.net.ServerSocket` được sử dụng bởi các ứng dụng máy chủ để tạo ra một một ứng dụng tại một cổng và lắng nghe các yêu cầu của máy khách.

Một đối tượng của lớp `ServerSocket` được tạo trên phía máy chủ và lắng nghe kết nối từ các máy khách. Đối tượng này luôn tồn tại trong một chương trình ứng dụng mạng đang chạy bằng giao thức TCP phía máy chủ.

5.4.1 Các phương thức tạo

`public` ***`ServerSocket`***`(int port) throws IOException`

Cố gắng tạo một `ServerSocket` bị ràng buộc vào port được chỉ định. Một ngoại lệ xảy ra nếu `port` đã bị ràng buộc bởi một ứng dụng khác.

`public` ***`ServerSocket`***`(int port, int backlog) throws IOException`

Tương tự như hàm tạo trước đó, tham số `backlog` xác định có bao nhiêu máy khách đến để lưu trữ trong một hàng đợi.

```
public ServerSocket(int port, int backlog, InetAddress
address) throws IOException
```

Tương tự như constructor trước đó, tham số `InetAddress` chỉ định địa chỉ IP cục bộ để ràng buộc. `InetAddress` được sử dụng cho các máy chủ có thể có nhiều địa chỉ IP, cho phép máy chủ xác định địa chỉ IP nào để chấp nhận yêu cầu của máy khách.

```
public ServerSocket() throws IOException
```

Tạo ra một `ServerSocket` không kết nối. Khi sử dụng constructor này, sử dụng phương thức `bind()` khi chúng ta muốn ràng buộc socket tới máy chủ.

5.4.2 Các phương thức khác

```
public Socket accept() throws IOException
```

Chờ cho một máy khách kết nối đến. Phương thức này ngăn chặn cho đến khi một máy trạm kết nối đến máy chủ trên cổng được chỉ định hoặc socket hết hạn, giả sử rằng giá trị thời gian đã được thiết lập bằng phương thức `setSoTimeout()`. Nếu không, phương thức này sẽ khóa lại vô thời hạn.

```
public int getLocalPort()
```

Trả về cổng mà socket của máy chủ lắng nghe. Phương thức này rất hữu ích nếu chúng ta truyền `0` như là số cổng trong một constructor và để cho máy chủ tìm thấy một cổng cho chúng ta.

```
public void setSoTimeout(int timeout)
```

Thiết lập giá trị thời gian chờ cho bao lâu socket của máy chủ chờ khách hàng trong suốt quá trình chấp nhận.

```
public void bind(SocketAddress host, int backlog)
```

Liên kết socket tới máy chủ và cổng được chỉ định trong đối tượng `SocketAddress`. Sử dụng phương thức này nếu chúng ta đã tạo ra các `ServerSocket` bằng cách sử dụng constructor không có đối số.

```
public void close()
```

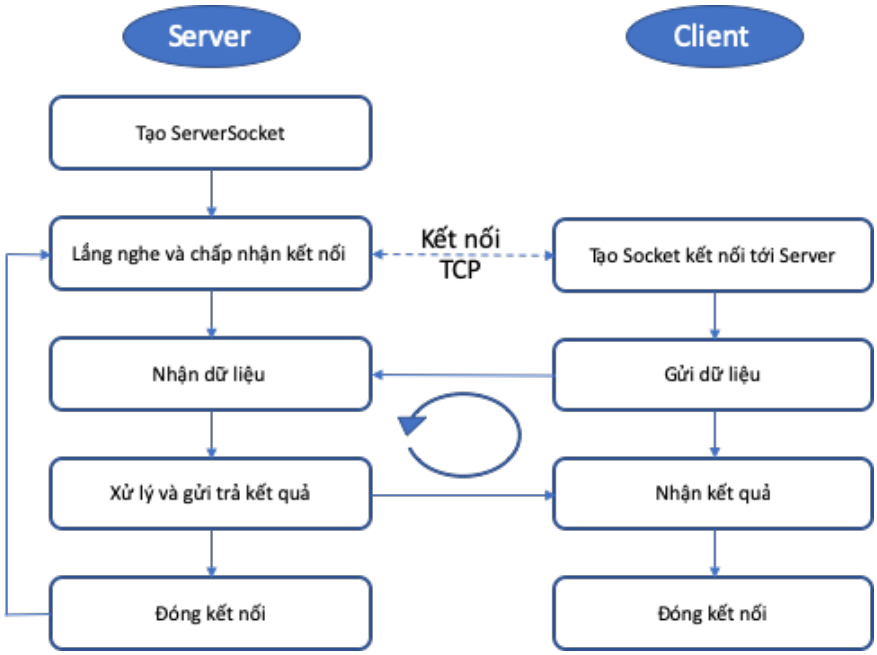
Đóng `ServerSocket`, ngừng phục vụ. Chúng ta ít khi sử dụng phương thức này vì `ServerSocket` thường luôn phục vụ phía máy chủ.

Khi `ServerSocket` gọi `accept()`, phương thức này sẽ không return cho đến khi một máy khách kết nối đến. Sau khi máy khách kết nối, `ServerSocket` tạo một

`Socket` mới trên một cổng không xác định và trả về một tham chiếu đến `Socket` mới này và thực hiện kết nối TCP giữa máy khách và máy chủ để có thể truyền tin.

5.5 Lập trình TCP bằng mô hình Client/Server

Trong mô hình lập trình TCP Client/Server với Java chúng ta sử dụng hai lớp `ServerSocket` và `Socket`. Lớp `ServerSocket` chỉ sử dụng ở phía Server trong khi lớp `Socket` sử dụng đồng thời ở phía Client và Server để trao đổi dữ liệu.



Hình 5.1: Mô hình Client/Server theo kỹ thuật lập trình với giao thức TCP

Quan sát hình trên chúng ta thấy rằng để tạo một ứng dụng mạng chạy bằng giao thức TCP chúng ta cần thiết lập hai ứng dụng riêng biệt: một ứng dụng Server và một ứng dụng cho Client.

Theo trình tự thời gian, ứng dụng phía máy chủ sẽ chạy trước và tạo ra một `ServerSocket` trên cổng `x` để lắng nghe các kết nối từ phía máy khách.

Máy khách sẽ tạo ra một `Socket` để kết nối tới máy chủ `hostid` qua cổng `x`.

Khi có yêu cầu kết nối, máy chủ sẽ chấp nhận kết nối bằng cách tạo ra một Socket qua phương thức `accept()`. Sau khi thiết lập kết nối máy chủ và máy khách có thể trao đổi dữ liệu thông qua các phương thức kiểm soát vào-ra của lớp `Socket`. Kết nối này sẽ tồn tại đến khi nào một trong hai bên hủy bỏ kết nối bằng cách đóng kết nối qua phương thức `close()` hoặc có sự cố về mạng.

5.6 Xử lý ngoại lệ trong lập trình mạng

Trong lập trình mạng nói chung, chúng ta thường xuyên gặp phải một số lỗi nhất định khi chạy chương trình. Có thể kể ra như lỗi xung đột cổng giữa các ứng dụng trên máy chủ, lỗi không kết nối được giữa các máy tính, lỗi không gửi/nhận dữ liệu được qua mạng…

Java định nghĩa một số ngoại lệ (Exception) để xử lý các sự cố này như: `IOException`, `UnknownHostException`...

Để xử lý các ngoại lệ này chúng ta có hai cách:

- Sử dụng cú pháp `try-catch`: chúng ta nên dùng cách này để chủ động trong việc xử lý lỗi. Khi có lỗi xảy ra, chúng ta có thể có biện pháp khắc phục hợp lý hoặc thông báo lỗi cho người dùng biết.
- Sử dụng cú pháp `throws` cho các phương thức: sử dụng cách này là chúng ta giao cho các lớp xử lý ngoại lệ của Java xử lý giúp. Cách này chỉ nên dùng với những lỗi đơn giản hoặc ít gặp phải.

5.7 Một số ví dụ

Ví dụ 5-1. *Viết chương trình kiểm tra một cổng trên máy chủ có đang hoạt động hay không.*

Cổng đang hoạt động được hiểu là cổng đang có một ứng dụng chạy trên đó. Có nghĩa là nó đang đóng vai trò là máy phục vụ trên cổng đó.

Việc thiết kế giao diện người dùng trong ví dụ này và các ví dụ sau được thực hiện trong phần mềm NetBeans. Chúng ta có thể sử dụng các công cụ khác để thiết kế hoặc tự sinh các đối tượng đồ họa bằng thư viện Swing và AWT.

Dùng Swing hoặc AWT thiết kế giao diện như sau, tên biến của các đối tượng đồ họa được chú thích ở cuối mũi tên.

Hình 5.2: Thiết kế giao diện kiểm tra cổng mạng

Xử lý sự kiện khi người dùng bấm nút *Kiểm tra* như sau:

```
private void btCheckActionPerformed(java.awt.event.ActionEvent evt) {
        String host = tfHost.getText(); //Máy chủ (host)
```

```
        int port = Integer.parseInt(tfPort.getText()); //Cổng (port)
        Socket sk = new Socket(); //Tạo socket
        try {
            sk.connect(new InetSocketAddress(host, port), 1000);
            JOptionPane.showMessageDialog(null, "Cổng "+port+" đang hoạt
động", "Trạng thái", 1);
            sk.close();
        } catch (IOException ex) {
            JOptionPane.showMessageDialog(null, "Cổng "+port+" đang không
hoạt động", "Trạng thái", 1);
        }
    }
```

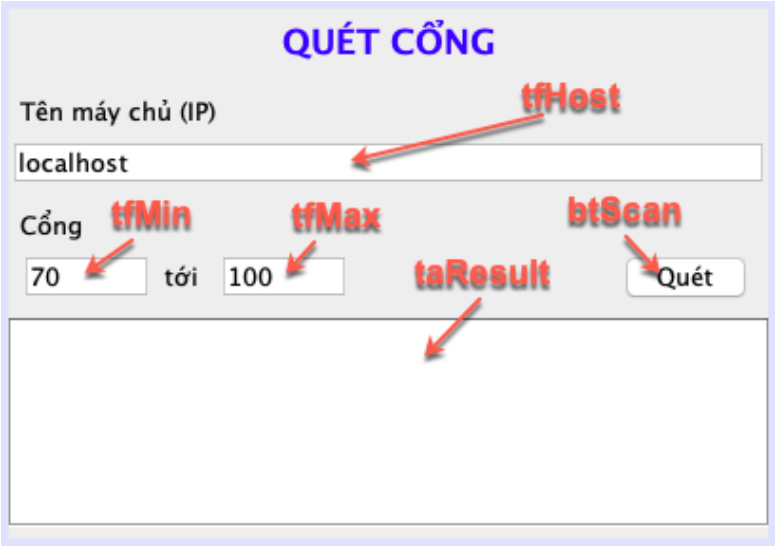
Kết quả chúng ta nhận được như sau:

Hình 5.3: Kết quả kiểm tra cổng mạng

Ví dụ 5-2. *Viết chương trình quét cổng trên máy chủ.*

Chương trình sẽ quét trong một phạm vi cổng nhất định trên máy chủ xem cổng nào đang hoạt động, cổng nào không hoạt động.

Thiết kế giao diện như hình dưới:



Hình 5.4: Thiết kế giao diện quét cổng mạng

Xử lý sự kiện khi người dùng bấm vào nút `btScan` như sau:

```
public void btScanActionPerformed(java.awt.event.ActionEvent evt) {
    String host = tfHost.getText();
    int min = Integer.parseInt(tfMin.getText());
    int max = Integer.parseInt(tfMax.getText());
    taResult.setText("");
    for (int i = min; i <= max; i++) {
        if (isActive(host,i,1000)) {
            taResult.append("Cổng "+i+" đang hoạt động\n");
        } else {
            taResult.append("Cổng "+i+" đang không hoạt động\n");
        }
    }
}

public boolean isActive(String host, int port, int timeout) {
    Socket sk = new Socket();
    try {
        sk.connect(new InetSocketAddress(host, port), timeout);
        sk.close();
        return true;
    } catch (IOException ex) {
        return false;
    }
}
```

Trong đoạn mã trên, phương thức `isActive(host, port, timeout)` dùng để kiểm tra xem cổng `port` trên máy chủ `host` có đang hoạt động hay không. Phương thức này sẽ trả về giá trị `true` nếu chương trình kết nối được với `host` qua cổng `port` trong thời gian chờ `timeout`, ngược lại nó sẽ trả về giá trị `false`.

Hình 5.5: Kết quả quét kiểm tra cổng mạng

***Ví dụ 5-3**. Viết chương trình theo mô hình Client/Server. Client gửi một xâu lên Server. Server chuyển xâu thành chữ in hoa rồi gửi trả lại cho Client.*

Bước 1: Lập trình phía Server.

- Tạo một project đặt tên là *TCP_Server*.

- Trong project này tạo một class và đặt tên là *Server*.

- Viết mã lệnh cho *Server* như sau:

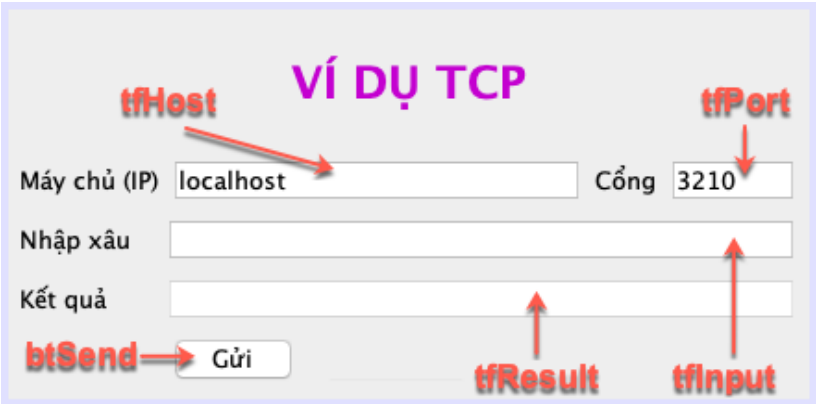
```
public class Server {
    private final int PORT = 3210;
    public static void main(String[] args) {
        new Server().run();
    }
    public void run() {
        try {
            ServerSocket server = new ServerSocket(PORT);
            System.out.println("Máy chủ đang chạy...");
            while (true) {
                Socket sk = server.accept();
                Scanner in = new Scanner(sk.getInputStream());
                PrintWriter out = new
                                  PrintWriter(sk.getOutputStream(),true);
                if (in.hasNextLine()) {
                    String inSt = in.nextLine();
                    String outSt = inSt.toUpperCase();
                    out.println(outSt);
                }
                sk.close();
            }
        } catch (IOException ex) {
            System.out.println("Không thể khởi chạy máy chủ!!!");;
        }
    }
}
```

Bước 2: Lập trình phía Client

- Tạo một project khác đặt tên là *TCP_Client*.

- Trong project này tạo một JFrame Form đặt tên là *Client*.

- Thiết kế giao diện như hình dưới:



Hình 5.6: Thiết kế giao diện Client xử lý xâu

- Xử lý sự kiện khi người dùng nhấn vào nút *Gửi*:

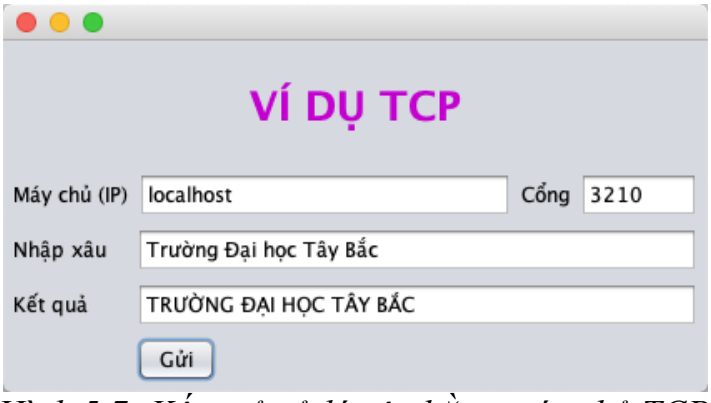
```
private void btSendActionPerformed(java.awt.event.ActionEvent evt) {
     String host = tfHost.getText();
     int port = Integer.parseInt(tfPort.getText());
     String inputStr = tfInput.getText();
     try {
        Socket sk = new Socket(host, port);
        Scanner in = new Scanner(sk.getInputStream());
     PrintWriter out = new PrintWriter(sk.getOutputStream(), true);
     out.println(inputStr);
        String serverStr = in.nextLine();
        tfResult.setText(serverStr);
        sk.close();
     } catch (IOException ex) {
            JOptionPane.showMessageDialog(null, "Không thể kết nối tới
máy chủ!!!", "Lỗi", 0);
     }
}
```

Bước 3: Chạy chương trình

- Chạy Server trước.

- Chạy Client sau, nhập dữ liệu và nhấn nút "Gửi".

- Kết quả như hình dưới.



Hình 5.7: Kết quả xử lý xâu bằng máy chủ TCP

Trên đây là một ví dụ đơn giản về kỹ thuật lập trình với giao thức TCP để xử lý xâu. Chúng ta thấy rằng việc xử lý xâu hoàn toàn ở phía Server, còn Client chỉ có nhiệm vụ gửi xâu và đợi kết quả. Việc xử lý xâu như thế nào là hoàn toàn do Server quyết định và một Server có thể xử lý yêu cầu của nhiều Client khác nhau.

Trong chương trình phía Server chúng ta thấy có vòng lặp `while(true)`, vòng lặp này không bao giờ dừng trừ khi chúng ta tự tắt chương trình. Việc có vòng lặp này đảm bảo rằng phía Server luôn chạy, nếu không có `while(true)` chương trình sẽ chỉ phục vụ một lần. Server có thể xử lý yêu cầu từ nhiều Client cùng một lúc.

Tuy đơn giản, nhưng ví dụ trên đã minh họa đầy đủ các bước thực hiện một giao tiếp mạng bằng giao thức TCP. Chúng ta có thể xử lý các yêu cầu phức tạp hơn ở phía Server hay gửi nhiều yêu cầu hơn ở phía Client. Đó là những kiến thức thuộc kỹ thuật lập trình, hoàn toàn có thể thực hiện được. Và để gửi nhiều dữ liệu hơn ở phía Client, chúng ta cùng xem xét ví dụ phía dưới.

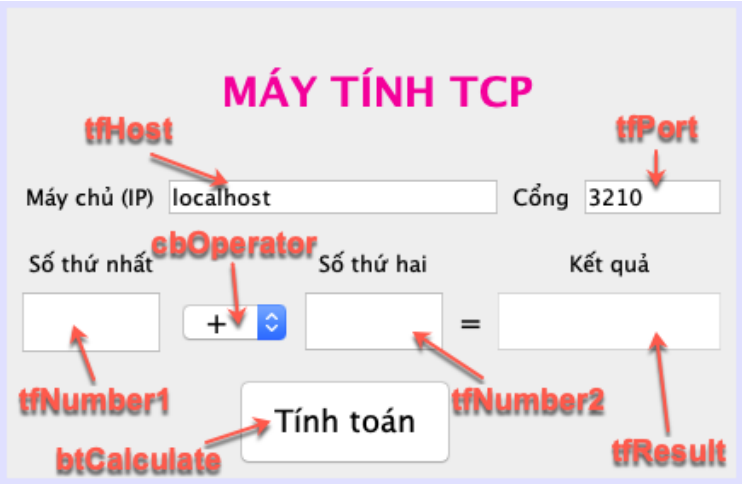
Ví dụ 5-4. *Viết chương trình theo mô hình Client/Server. Client gửi lên Server hai số thực và một trong bốn phép toán: cộng, trừ, nhân, chia. Server xử lý tính toán theo yêu cầu và gửi trả kết quả.*

Bước 1: Lập trình phía Client

- Tạo project đặt tên là *TCP_Calculator_Client*.

- Trong project này tạo một JFrame Form đặt tên là *Client*.

- Thiết kế giao diện cho *Client* như sau:



Hình 5.8: Thiết kế giao diện Client xử lý số

- Xử lý sự kiện người dùng bấm vào nút *Tính toán* như sau:

```
private void btCalculateActionPerformed(java.awt.event.ActionEvent evt) {
    try {
        String host = tfHost.getText();
        int port = Integer.parseInt(tfPort.getText());
        double n1 = Double.parseDouble(tfNumber1.getText());
        double n2 = Double.parseDouble(tfNumber2.getText());
        String operator = cbOperator.getSelectedItem().toString();
        Socket sk = new Socket(host,port);
        Scanner in = new Scanner(sk.getInputStream());
        PrintWriter out = new PrintWriter(sk.getOutputStream(),true);
        out.println(n1);
        out.println(n2);
        out.println(operator);
        tfResult.setText(in.nextLine());
        sk.close();
    } catch (IOException ex) {
```

```
        JOptionPane.showMessageDialog(null, "Không thể kết nối tới máy
chủ!!!", "Lỗi", 0);
    } catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(null, "Vui lòng nhập dữ liệu hợp
lệ!!!", "Lỗi", 0);
    }
}
```

Bước 2: Lập trình phía Server

- Tạo một project đặt tên là *TCP_Calculator_Server*.

- Trong project này tạo một class và đặt tên là *Server*.

- Viết mã lệnh cho *Server* như sau:

```
public class Server {
    private final int PORT = 3210;
    public static void main(String[] args) {
        new Server().run();
    }
    public void run() {
        try {
            ServerSocket server = new ServerSocket(PORT);
            System.out.println("Máy chủ đang chạy...");
            while (true) {
                Socket sk = server.accept();
                Scanner in = new Scanner(sk.getInputStream());
                PrintWriter out = new
PrintWriter(sk.getOutputStream(),true);
                double n1 = in.nextDouble();
                double n2 = in.nextDouble();
                String operator = in.next();
                String result = "";
                switch (operator) {
                    case "+":
                        result = (n1+n2) + "";
                        break;
                    case "-":
                        result = (n1-n2) + "";
                        break;
                    case "*":
                        result = (n1*n2) + "";
                        break;
                    case "/":
                        DecimalFormat f = new DecimalFormat("#.##");
                        result = f.format(n1/n2) + "";
                        break;
                }
                out.println(result);
                sk.close();
            }
        } catch (IOException ex) {
            System.out.println("Không thể khởi chạy máy chủ!!!");
        }
```

```
    }
}
```

Bước 3: Chạy chương trình

- Chạy *Server* trước.

- Chạy *Client* sau, nhập dữ liệu và nhấn nút *Tính toán*.

- Kết quả như hình dưới.

Hình 5.9: Kết quả xử lý số bằng máy chủ TCP

Trong đoạn mã phía Client, chúng ta thấy việc gửi dữ liệu (`n1`, `n2`, `operator`) lên Server được thực hiện nối tiếp nhau bằng ba câu lệnh `println()`. Như là một giao ước, phía bên Server cũng sẽ phải nhận liên tiếp ba giá trị này.

Trong kỹ thuật lập trình mạng, sau khi thiết lập kết nối TCP các máy tính có thể thực hiện nhiều lần việc gửi và nhận dữ liệu. Tuy nhiên, chúng ta cần thiết lập quy tắc trao đổi dữ liệu để sao cho một bên gửi thì bên kia nhận dữ liệu. Việc gửi-nhận này cần phải nhịp nhàng, chính xác.

Thêm vào đó, thay vì phải gửi liên tiếp ba lần cho ba dữ liệu, chúng ta có thể nối 3 dữ liệu này thành một xâu duy nhất và gửi đi một lần. Phía Server chia tách xâu và xác định lại các dữ liệu cần thiết. Cấu trúc của xâu ghép phải được Client và Server thống nhất với nhau. Giả sử, ở ví dụ trên thay vì thực hiện ghi ba lần liên tiếp lên dòng ra các dữ liệu `n1`, `n2`, `operator` ta chỉ cần nối chúng thành một xâu ngăn cách bởi kí tự đặc biệt và gửi tới máy chủ xâu đó: "`n1@n2@operator`".

CÂU HỎI, BÀI TẬP VẬN DỤNG:

1. Lớp *Socket* dùng để làm gì? Liệt kê các phương thức của lớp *Socket*?
2. Lớp *ServerSocket* dùng để làm gì? Liệt kê các phương thức của lớp *ServerSocket*?

3. Trình bày kỹ thuật lập trình với giao thức TCP bằng mô hình Client/Server?
4. Sử dụng kỹ thuật lập trình với giao thức TCP, viết chương trình Java theo mô hình Client/Server xử lý xâu như sau:
   - Tính số từ của xâu.
   - Tìm xâu đảo ngược của xâu. Ví dụ: "ABCD" → "DCBA".
5. Sử dụng kỹ thuật lập trình với giao thức TCP, viết chương trình Java theo mô hình Client/Server thực hiện tính toán như sau:
   - Chuyển một số sang hệ Hexa-Decimal (hệ thập lục phân).
   - Kiểm tra xem số đã cho có phải số hoàn hảo không?

   (N là số hoàn hảo nếu N có tổng các ước số bằng chính nó - trừ ước là N)
6. Sử dụng kỹ thuật lập trình với giao thức, viết chương trình Java theo mô hình Client/Server thực hiện tính toán như sau:
   - Tìm ước số chung lớn nhất của A và B.
   - Tìm xem có số nào trong hai số là số nguyên tố không?
7. Viết chương trình chat đơn giản trong mạng LAN sử dụng giao thức TCP?
8. Viết chương trình trò chơi TicTacToe giữa 2 người trên 2 máy tính khác nhau sử dụng giao thức TCP?
9. Viết chương trình trò chơi đuổi hình bắt chữ bằng giao thức TCP với hình ảnh và dữ liệu câu hỏi nằm trên máy chủ?
10. Viết chương trình thi trắc nghiệm với bộ câu hỏi nằm trong cơ sở dữ liệu trên máy chủ sử dụng giao thức TCP?

CHƯƠNG 6. LẬP TRÌNH VỚI GIAO THỨC UDP

6.1 Khái niệm chung

UDP là viết tắt của cụm từ User Datagram Protocol. UDP là một phần của bộ giao thức Internet được sử dụng bởi các chương trình chạy trên các máy tính khác nhau trên mạng. Không giống như TCP, UDP được sử dụng để gửi các gói tin ngắn gọi là datagram, cho phép truyền nhanh hơn. Tuy nhiên, UDP không cung cấp kiểm tra lỗi nên không đảm bảo toàn vẹn dữ liệu.

Giao thức UDP hoạt động tương tự như TCP nhưng nó không tạo ra một kết nối giữa các máy tính và không cung cấp kiểm tra lỗi khi truyền gói tin. Khi một ứng dụng sử dụng UDP, các gói tin chỉ được gửi đến người nhận. Người gửi không đợi để đảm bảo người nhận có nhận được gói tin hay không, mà tiếp tục gửi các gói tiếp theo. Nếu người nhận bỏ lỡ một vài gói tin UDP, gói tin đó bị mất vì người gửi sẽ không gửi lại chúng. Điều này có nghĩa là các thiết bị có thể giao tiếp nhanh hơn.

UDP được sử dụng khi tốc độ được ưu tiên và sửa lỗi là không cần thiết. UDP thường được sử dụng cho phát sóng trực tuyến và trò chơi trực tuyến. Ví dụ như khi xem một luồng video trực tiếp, thường được phát bằng UDP thay vì TCP. Máy chủ chỉ gửi một luồng UDP liên tục tới các máy tính đang xem. Nếu bị mất kết nối trong vài giây, video có thể bị ngưng hoặc lag trong chốc lát và sau đó phát tiếp phần hiện tại. Nếu bị mất gói tin nhỏ, video hoặc âm thanh có thể bị méo mó một chút khi video tiếp tục phát mà không có dữ liệu bị mất.

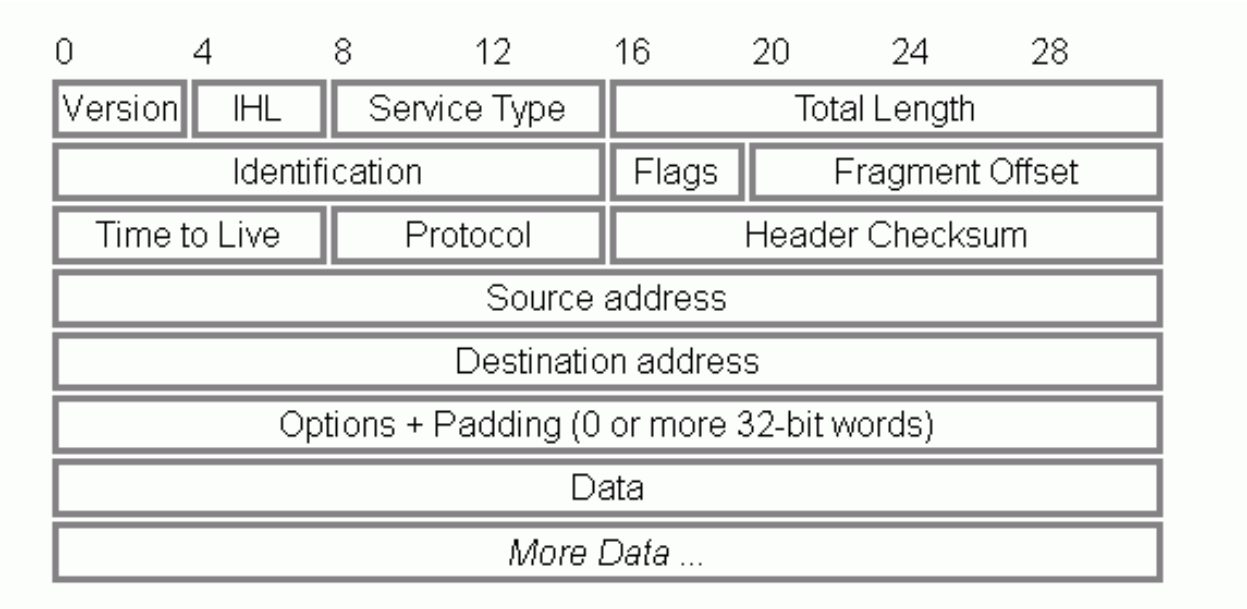
Điều này hoạt động tương tự trong các trò chơi trực tuyến. Nếu chúng ta bỏ lỡ một số gói UDP, các nhân vật của người chơi có thể xuất hiện trên bản đồ ở vị trí khác khi nhận được các gói UDP mới hơn.

Sự khác nhau giữa TCP và UDP thường được minh họa bằng sự khác nhau giữa hệ thống điện thoại và hệ thống bưu chính. TCP giống với hệ thống điện thoại. Khi chúng ta gọi một số điện thoại nào đó, người đó cần đồng ý kết nối bằng cách nhấc điện thoại (trả lời cuộc gọi). Nếu điện thoại bận hoặc không có ai nhấc máy, chúng ta không liên lạc được. UDP thì giống như hệ thống bưu chính. Bạn gửi thư tay tới một địa chỉ cố định thông qua hệ thống bưu chính. Thư có thể tới nơi hoặc không tới nơi nhưng bạn không thể biết chắc được điều này. Bạn có thể gửi nhiều thư đi cho người nhận, đánh số chúng và yêu cầu họ gửi thư lại xác nhận xem thư nào tới, thư nào chưa tới. Bạn và người nhận phải thực hiện điều này chứ hệ thống bưu chính không thực hiện cho bạn.

Cả hệ thống điện thoại và hệ thống bưu chính hiện vẫn đang hoạt động song song cho những mục đích khác nhau. Cũng giống như vậy, TCP và UDP đều có mục

đích sử dụng riêng. Chúng ta không thể nói rằng cái nào tốt hơn cái nào. Thay vào đó, với từng ứng dụng cụ thể hay nói chính xác hơn là với từng yêu cầu cụ thể chúng ta sử dụng TCP hay UDP.

Trong Java, cả UDP Server và UDP Client đều sử dụng các đối tượng của lớp `java.net.DatagramSocket` để giao tiếp và `java.net.DatagramPacket` là lớp được sử dụng để đóng gói dữ liệu để gửi đi và nhận dữ liệu dưới dạng packet. Số lượng byte lớn nhất có thể gửi thông qua UDP là *65507* byte cho một lần. Mặc dù vậy, trên các ứng dụng thực chúng ta ít khi sử dụng hết độ dài này mà thường trong khoảng 8.192 bytes (8K). Và khi tiến hành gửi/nhận dữ liệu chúng ta cũng dùng kích thước bé hơn nhiều.



Hình 6.1: Cấu tạo của DatagramPacket

Để gửi dữ liệu, chúng ta cần chuyển nó về kiểu `byte[]`, đưa dữ liệu vào `DatagramPacket` và gửi nó qua một `DatagramSocket`. Để nhận dữ liệu, ta nhận một `DatagramPacket` thông qua một `DatagramSocket` và tách lấy dữ liệu từ packet. Dữ liệu nhận được là kiểu `byte[]`, do đó chúng ta cần chuyển nó về kiểu dữ liệu thích hợp để lấy được thông tin cần thiết. `DatagramPacket` cũng có các phương thức giúp đọc thông tin liên quan tới máy gửi như địa chỉ mạng, cổng,…

Việc sử dụng hai lớp `DatagramPacket` và `DatagramSocket` tương phản với TCP sử dụng lớp `Socket` và `ServerSocket`. Khác biệt thứ nhất giữa TCP và UDP là UDP không thực hiện một kết nối giữa hai máy tính khi gửi/nhận dữ liệu. Một socket có thể gửi và nhận dữ liệu từ nhiều máy tính khác nhau chứ không phụ thuộc vào một kết nối như TCP. Thứ hai, TCP socket sử dụng một kết nối để điều khiển các luồng dữ liệu. Với TCP, chúng ta gửi/nhận thông qua các dòng vào/dòng ra của socket. UDP thì không như vậy, chúng ta làm việc với các datagram packet riêng lẻ.

Các packet không liên quan với nhau và khi nhận chúng ta không thể biết packet nào được gửi trước, packet nào được gửi sau.

6.2 Lớp DatagramSocket

Không giống như TCP, đối với UDP thì cả bên nhận và bên gửi sẽ cùng sử dụng lớp `DatagramSocket` để giao tiếp với nhau. Một số phương thức chính:

```
public DatagramSocket() throws SocketException
```

Hàm khởi tạo UDP socket cho Client. Khởi tạo UDP socket và chưa chỉ định cổng cụ thể, dùng các cổng còn trống của hệ thống.

```
public DatagramSocket(int port) throws SocketException
```

Hàm khởi tạo UDP socket cho Server. Khởi tạo UDP socket và ràng buộc nó vào một `port` cụ thể được chỉ ra.

```
public synchronized void setSoTimeout(int timeout) throws SocketException
```

Phương thức thiết lập thời gian chờ cho `DatagramSocket`. Hết thời gian chờ này không có phản hồi từ phía máy gửi, socket sẽ tự hủy.

```
public void send(DatagramPacket p) throws IOException
```

Gửi `DatagramPacket` đến host nhận. `DatagramPacket` chứa dữ liệu cần gửi, độ dài dữ liệu, địa chỉ IP và số hiệu port của host sẽ nhận.

```
public void receive(DatagramPacket p) throws IOException
```

Thực hiện nhận về packet từ `DatagramSocket`. Khi phương thức này được gọi thành công, `buf` của `DatagramPacket` sẽ chứa nội dung dữ liệu nhận được. Đồng thời `DatagramPacket` còn chứa thông tin về địa chỉ IP và port của bên gửi. Phương thức này khi gọi sẽ bị block cho đến khi có 1 `DatagramPacket` được nhận.

```
void bind(SocketAddress addr) throws SocketException
```

Ràng buộc `DatagramSocket` vào một địa chỉ mạng cụ thể (IP và port). Phương thức này thường được dùng khi tạo `DatagramSocket` bằng phương thức thứ nhất, tức là không chỉ định rõ cổng cụ thể hoặc là muốn thay đổi địa chỉ mạng cho UDP socket.

```
public void close()
```

Đóng `DatagramSocket`.

6.3 Lớp DatagramPacket

Lớp `DatagramPacket` trong Java dùng để đóng gói dữ liệu để gửi đi, đồng thời cũng dùng để nhận dữ liệu từ `DatagramSocket`. Một số phương thức chính:

```
public DatagramPacket(byte buf[], int length)
```

Khởi tạo một `DatagramPacket` dùng để nhận 1 packet có độ dài là `length` nhỏ hơn hoặc bằng `buf.length`; `buf[]` là vùng nhớ đệm dùng để lưu dữ liệu sắp nhận; `length` là số lượng byte lớn nhất được dùng để nhận dữ liệu.

```
public DatagramPacket(byte buf[], int length, InetAddress
address, int port)
```

Khởi tạo một `DatagramPacket` để gửi 1 packet có độ dài là `length` đến cổng có số hiệu `port` trên host cụ thể được chỉ ra trong `address`; `buf[]` chính là dữ liệu muốn gửi.

```
public InetAddress getAddress()
```

Trả về địa chỉ IP của host đã gởi packet hoặc host sẽ nhận packet.

```
public int getPort()
```

Trả về giá trị port của host đã gởi packet hoặc host mà packet sẽ được gởi đến.

```
public byte[] getData()
```

Trả về nội dung dữ liệu trong `DatagramPacket`

```
public int getLength()
```

Trả về độ dài của dữ liệu trong `DatagramPacket.`

```
public synchronized int getOffset()
```

Trả về *offset* (độ lệch) của dữ liệu trong `DatagramPacket`.

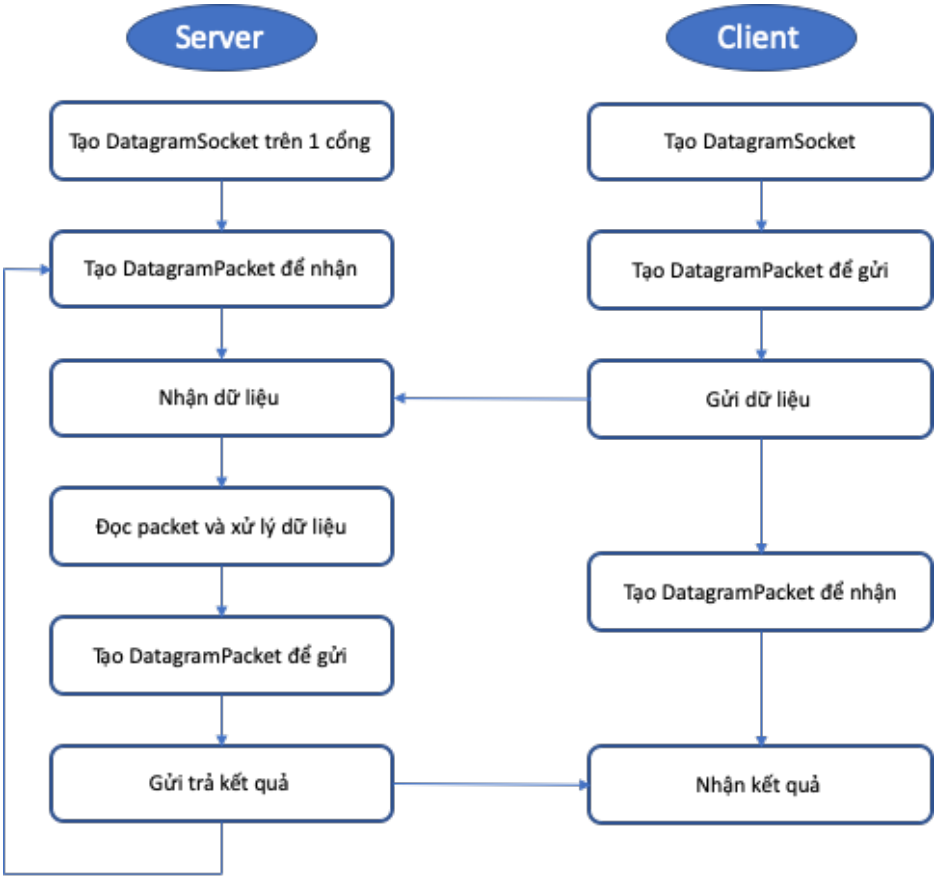
6.4 Lập trình UDP theo mô hình Client/Server

Trong mô hình lập trình UDP Client/Server đều sử dụng hai lớp `DatagramSocket` và `DatagramPacket` cho cả phía Client và Server. Tuy nhiên cách sử dụng hai lớp này ở phía Client và Server là khác nhau trong các constructor.

Quan sát mô hình trên *Hinh 6.2*, ta thấy rằng để tạo một giao tiếp bằng UDP, phía máy chủ sẽ tạo ra một socket và ràng buộc trên một cổng nhất định nào đó.

Muốn gửi dữ liệu (hoặc yêu cầu) thì phải biết trước các thông tin liên quan như địa chỉ mạng, cổng kết nối của máy nhận. Máy gửi sẽ tạo ra một socket dùng cho việc gửi dữ liệu. Máy gửi tạo ra một `DatagramPacket` với các thông tin như dữ liệu,

chiều dài dữ liệu, địa chỉ mạng, cổng kết nối của máy nhận. Chúng ta sẽ gọi tới socket đã tạo và gửi packet đi.



Hình 6.2: Mô hình Client/Server theo kỹ thuật lập trình với giao thức UDP

Đoạn chương trình để gửi dữ liệu bằng giao thức UDP:

```
DatagramSocket socket = new DatagramSocket();
byte[] data = inputStr.getBytes();
InetAddress host = InetAddress.getByName("localhost");
int port = 3210;
DatagramPacket sPacket = new DatagramPacket(data,data.length,host,port);
socket.send(sPacket);
```

Phía máy nhận muốn nhận dữ liệu thì sẽ tạo ra một socket để gửi/nhận dữ liệu. Nếu đã có socket rồi thì có thể sử dụng lại. Sau đó, máy nhận tạo ra một packet với thông tin về biến nhớ đệm và độ dài dữ liệu. Sau đó, chúng ta sẽ dùng socket để nhận dữ liệu.

Đoạn chương trình để gửi dữ liệu bằng giao thức UDP:

```
DatagramSocket socket = new DatagramSocket();
byte[] buffer = new byte[65507];
DatagramPacket rPacket = new DatagramPacket(buffer,buffer.length);
socket.receive(rPacket);
```

Trong đoạn mã trên, *65507* được dùng làm kích thước của dữ liệu nhận. Đây là kích thước lớn nhất có thể dùng. Tuy nhiên, trên thực tế khi dùng ta có thể sử dụng số bé hơn.

Để đọc thông tin của packet nhận được, ta có đoạn mã sau:

```
InetAddress host = rPacket.getAddress(); //Địa chỉ mạng của máy gửi
int port = rPacket.getPort(); //Cổng của máy gửi
byte[] data = rPacket.getData(); //Dữ liệu nhận được
//Tạo xâu từ dữ liệu nhận được
String str = new String(data,rPacket.getOffset(),rPacket.getLength());
```

Thông tin đọc được từ phía máy gửi có thể dùng để gửi lại phản hồi sau này. Nói một cách chính xác thì thông tin địa chỉ mạng và cổng sẽ dùng để đóng gói thông tin qua một `DatagramPacket` dùng để gửi phản hồi cho máy vừa gửi thông tin. Điều này cũng giống như việc gửi thư, khi nhận được thư ta thường đọc thông tin của người gửi (nếu không biết trước) trên phong bì thư để gửi lại thư phản hồi.

6.5 Một số ví dụ

Ví dụ 6-1. *Viết chương trình UDP kiểm tra một xâu có phải là xâu đối xứng hay không. Client sẽ gửi lên Server một xâu. Server sẽ gửi trả kết quả kiểm tra tính đối xứng của xâu.*

Bước 1: Lập trình phía Server.

- Tạo một project đặt tên là *UDP_Server*.

- Trong project này tạo một class và đặt tên là *Server*.

- Viết mã lệnh cho *Server* như sau:

```
public class Server {
    private final int PORT = 3210;
    public static void main(String[] args) {
        new Server().run();
    }
    public void run() {
        try {
            DatagramSocket server = new DatagramSocket(PORT);
            System.out.println("Máy chủ đang chạy...");
            while (true) {
                byte[] buffer = new byte[65507];
                DatagramPacket rPacket = new
DatagramPacket(buffer,buffer.length);
                server.receive(rPacket);
                InetAddress host = rPacket.getAddress();
                int port = rPacket.getPort();
                String inputStr = new
String(rPacket.getData(),rPacket.getOffset(),rPacket.getLength());//Xau
nhan duoc
```

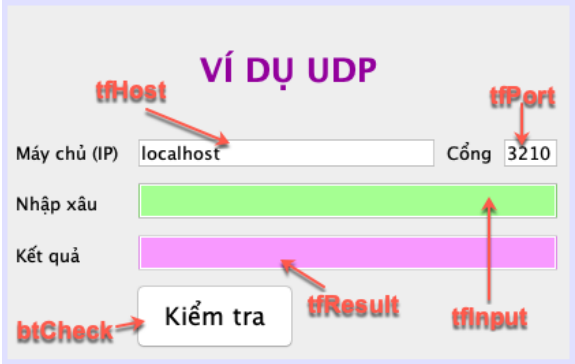
```
                String reverseStr = "";
                for (int i = inputStr.length()-1; i >= 0; i--) {
                    reverseStr += inputStr.charAt(i);
                }
                String result = (reverseStr.equals(inputStr)) ? "1" :
"0";
                byte[] data = result.getBytes();
                DatagramPacket sPacket = new
DatagramPacket(data,data.length,host,port);
                server.send(sPacket);
            }
        } catch (SocketException ex) {
            System.out.println("Không thể khởi chạy máy chủ!!!");
        } catch (IOException ex) {
            System.out.println("Không thể gửi/nhận dữ liệu!!!");
        }
    }
}
```

Bước 2: Lập trình phía *Client*

- Tạo một project khác đặt tên là *UDP_Client*.

- Trong project này tạo một JFrame Form đặt tên là *Client*.

- Thiết kế giao diện như hình dưới:



Hình 6.3: Thiết kế giao diện xử lý xâu bằng UDP

- Xử lý sự kiện khi người dùng nhấn vào nút *Kiểm tra*:

```
private void btCheckActionPerformed(java.awt.event.ActionEvent evt) {
     try {
          InetAddress host = InetAddress.getByName(tfHost.getText());
          int port = Integer.parseInt(tfPort.getText());
          String inputStr = tfInput.getText();
          DatagramSocket sk = new DatagramSocket();
         byte[] data = inputStr.getBytes();
         DatagramPacket sPacket = new
DatagramPacket(data,data.length,host,port);
         sk.send(sPacket);
         byte[] buffer = new byte[65507];
         DatagramPacket rPacket = new
DatagramPacket(buffer,buffer.length);
         sk.receive(rPacket);
```

```
        String result = new
String(rPacket.getData(),rPacket.getOffset(),rPacket.getLength());
        if (result.equals("1")) {
          tfResult.setText("Xâu đã nhập là đối xứng");
        } else {
          tfResult.setText("Xâu đã nhập không đối xứng");
        }
        sk.close();
    } catch (UnknownHostException ex) {
     JOptionPane.showMessageDialog(null, "Không kết nối được tới máy
chủ!!!", "Lỗi", 0);
    } catch (SocketException ex) {
     JOptionPane.showMessageDialog(null, "Không thể tạo socket!!!",
"Lỗi", 0);
    } catch (IOException ex) {
     JOptionPane.showMessageDialog(null, "Không thể gửi/nhận dữ liệu!!!",
"Lỗi", 0);
    }
}
```

Bước 3: Chạy chương trình

- Chạy *Server* trước.

- Chạy *Client* sau, nhập dữ liệu và nhấn nút *Kiểm tra*.

- Kết quả như hình dưới.

Hình 6.4: Kết quả xử lý xâu bằng máy chủ UDP

Trên đây là một ví dụ đơn giản về kỹ thuật lập trình với giao thức UDP để xử lý xâu. So với các ví dụ về TCP, rõ ràng chúng ta chương trình không tạo một kết nối giữa Server và Client. Việc giao tiếp giữa chúng được thực hiện thông qua các `DatagramSocket`. Dữ liệu được gửi/nhận bằng cách tạo ra các `DatagramPacket` với các phương thức tạo dùng để gửi và để nhận. Các `DatagramPacket` này được gửi/nhận thông qua các `DatagramSocket`.

Chúng ta có thể làm lại các ví dụ ở chương trước (TCP) với kỹ thuật trao đổi dữ liệu Client/Server theo ví dụ trên.

CÂU HỎI, BÀI TẬP VẬN DỤNG:

1. Lớp *DatagramSocket* dùng để làm gì? Liệt kê các phương thức của lớp *DatagramSocket*?
2. Lớp *DatagramPacket* dùng để làm gì? Liệt kê các phương thức của lớp *DatagramPacket*?
3. Trình bày kỹ thuật lập trình với giao thức UDP bằng mô hình Client/Server?
4. Phân biệt UDP với TCP và kỹ thuật lập trình với hai giao thức này?
5. Sử dụng kỹ thuật lập trình với giao thức UDP, viết chương trình Java theo mô hình Client/Server xử lý xâu như sau:
   - Tính số từ của xâu.
   - Tìm xâu đảo ngược của xâu. Ví dụ: "ABCD" → "DCBA".
6. Sử dụng kỹ thuật lập trình với giao thức UDP, viết chương trình Java theo mô hình Client/Server thực hiện tính toán như sau:
   - Chuyển một số sang hệ Hexa-Decimal (hệ thập lục phân).
   - Kiểm tra xem số đã cho có phải số hoàn hảo không?

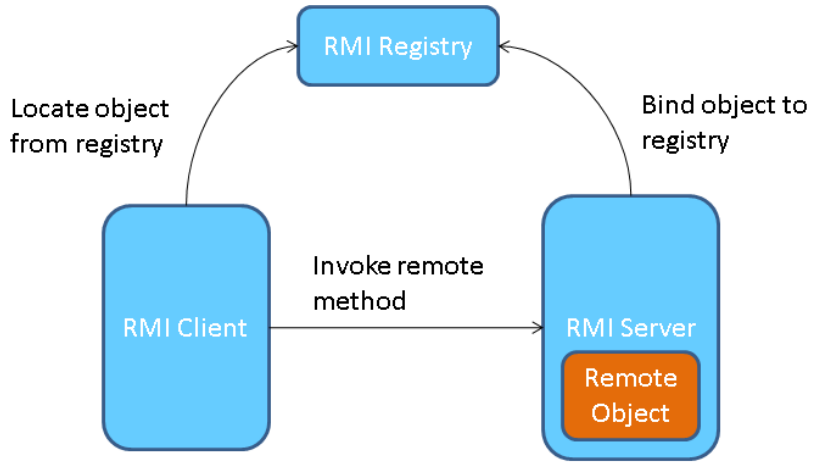
   (N là số hoàn hảo nếu N có tổng các ước số bằng chính nó - trừ ước là N)
7. Sử dụng kỹ thuật lập trình với giao thức UDP, viết chương trình Java theo mô hình Client/Server thực hiện tính toán như sau:
   - Tìm ước số chung lớn nhất của A và B.
   - Tìm xem có số nào trong hai số là số nguyên tố không?
8. Viết chương trình chat đơn giản trong mạng LAN sử dụng giao thức UDP?
9. Viết chương trình chuyển đổi mệnh giá các loại tiền tệ với tỉ giá được lưu trữ ở máy chủ, sử dụng giao thức UDP?
10. Viết chương trình cung cấp thông tin thời tiết cho một số thành phố tại Việt Nam với dữ liệu thời tiết được lưu trữ ở máy chủ, sử dụng giao thức UDP?

CHƯƠNG 7. KỸ THUẬT LẬP TRÌNH PHÂN TÁN RMI

7.1 Khái niệm chung

Lập trình đối tượng phân tán là một vấn đề hấp dẫn của công nghệ phân tán phần mềm ngày này. Java là ngôn ngữ đi tiên phong với RMI (Remote Method Invocation), một kỹ thuật cài đặt các đối tượng phân tán vô cùng hiệu quả và linh hoạt.

Thông thường các chương trình của chúng ta được viết dưới dạng thủ tục - hàm và việc các hàm gọi lẫn nhau và truyền tham số chỉ xảy ra ở máy cục bộ. Kỹ thuật RMI - mang ý nghĩa là triệu gọi phương thức từ xa là cách thức giao tiếp giữa các đối tượng trong Java có mã lệnh cài đặt nằm ở trên các máy khác nhau có thể triệu gọi lẫn nhau.



Hình 7.1: Mô hình RMI tổng quát

7.2 Kỹ thuật lập trình RMI theo mô hình Client/Server

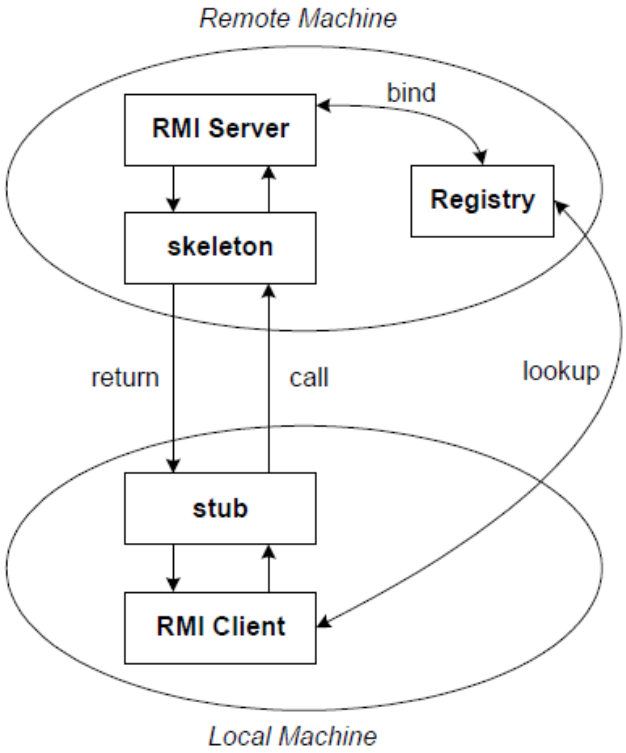
Để giải quyết một số vấn đề trong việc truyền thông giữa Client ⇔ Server. RMI không gọi trực tiếp mà thông qua lớp trung gian. Lớp này tồn tại ở cả hai phía Client và Server:

- Lớp ở Client gọi là *Stub*
- Lớp ở máy Server gọi là *Skel* (Skeleton)

Các đặc tính của RMI:

- RMI là mô hình đối tượng phân tán của Java, nó giúp cho việc truyền thông giữa các đối tượng phân tán được dễ dàng hơn.
- RMI là API bậc cao được xây dựng dựa trên lập trình socket.

- RMI không những cho phép chúng ta truyền dữ liệu giữa các đối tượng trên các hệ thống máy tính khác nhau và còn gọi được các phương thức trong các đối tượng ở xa.
- Việc truyền dữ liệu giữa các máy khác nhau được sử lý một cách trong suốt bởi máy ảo Java (Java Virtual Machine).
- RMI cung cấp cơ chế callback, nó cho phép Server triệu gọi các phương thức ở Client.



Hình 7.2: Kiến trúc cơ bản của RMI

Kiến trúc của RMI:

- Remote interface: Nên extend từ `java.rmi.Remote`. Nó khai báo tất cả các phương thức mà Client có thể triệu gọi. Tất cả các phương thức trong interface này nên `throws RemoteException`.
- Remote implementation: Được thực thi từ Remote interface và mở rộng từ `UnicastRemoteObject`. Triển khai các phương thức được khai báo trong interface tại đây. Nó là một Remote Object thực sự. Phát sinh hai lớp trung gian Stub và Skeleton.
- Server class bao gồm:
  - Các class được hiện thực trên Server.
  - RMI registry: Bộ đăng kí này sẽ đăng kí một Remote object với Naming Registry. Giúp các Remote object được chấp nhận khi gọi các phương thức từ xa.

- Client class: Truy vấn trên tên Remote object trên RMI registry, thông qua Stub để gọi các phương thức trên Server.

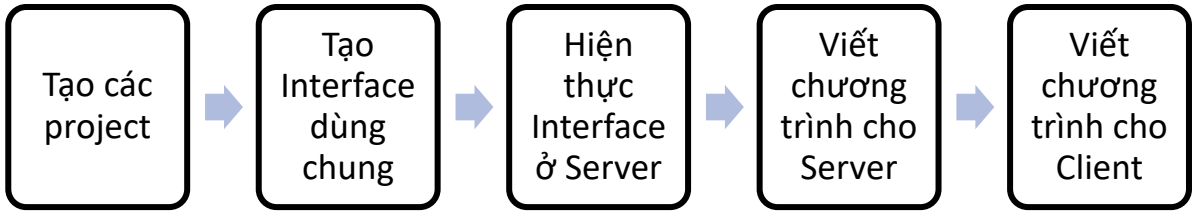
Truyền tin trong RMI:

- RMI sử dụng lớp trung gian để truyền tin Skeleton và Stub.
- Lớp Stub dùng ở Client.
- Lớp Skeleton dùng ở Server.
- Java tạo ra các lớp trung gian.
- RMI sử dụng các TCP Socket.

Cách thức hoạt động của RMI:

- Server RMI phải đăng ký với 1 dịch vụ tra tìm và đăng ký tên miền.
- Sau khi Server được đăng ký, nó sẽ chờ các yêu cầu của RMI client.
- Các ClientRMI sẽ gửi thông điệp RMI để gọi một phương thức trên một đối tượng từ xa.
- Ứng dụng Client yêu cầu một tên dịch vụ cụ thể và nhận một URL trỏ tới tài nguyên từ xa.

Mô hình lập trình phân tán RMI:



Hình 7.3: Các bước lập trình theo kỹ thuật RMI

- Bước 1: Tạo project cho Client và Server.
- Bước 2: Tạo Interface dùng chung cho cả Client và Server.
- Bước 3: Hiện thực Interface ở Server.
- Bước 4: Viết chương trình phía Server.
- Bước 5: Viết chương trình phía Client.

7.3 Một số ví dụ

Ví dụ 7-1*. Viết chương trình liệt kê các số nguyên tố từ 1 tới N, với N là một số nguyên dương, sử dụng kỹ thuật lập trình RMI. Phương thức kiểm tra số nguyên tố được triệu gọi từ xa.*

Bước 1: Tạo 2 project *RMI_Prime_Client* và *RMI_Prime_Server*.

Bước 2: Trong project *RMI_Prime_Server* tạo một package đặt tên là *Core*. Trong package này tạo một interface đặt tên là `PrimeInterface` như sau:

```
public interface PrimeInterface extends Remote{
    public boolean isPrime(int x) throws RemoteException;
}
```

Chú ý rằng trong kỹ thuật lập trình RMI các Interface phải kế thừa lớp `Remote`, các phương thức của nó phải `throws RemoteException`.

Phương thức `isPrime(int x)` dùng để kiểm tra một số `x` có phải là số nguyên tố hay không. Phương thức này chưa được hiện thực mà mới chỉ khai báo.

Sao chép package *Core* sang project *RMI_Prime_Client* (bao gồm cả `PrimeInterface`).

Bước 3: Hiện thực `PrimeInterface` phía Server. Trong project *RMI_Prime_Server* tạo một package mới đặt tên là *RMI*. Trong package này tạo một lớp mới đặt tên là `Prime`.

Hiện thực hóa interface trong lớp này như sau:

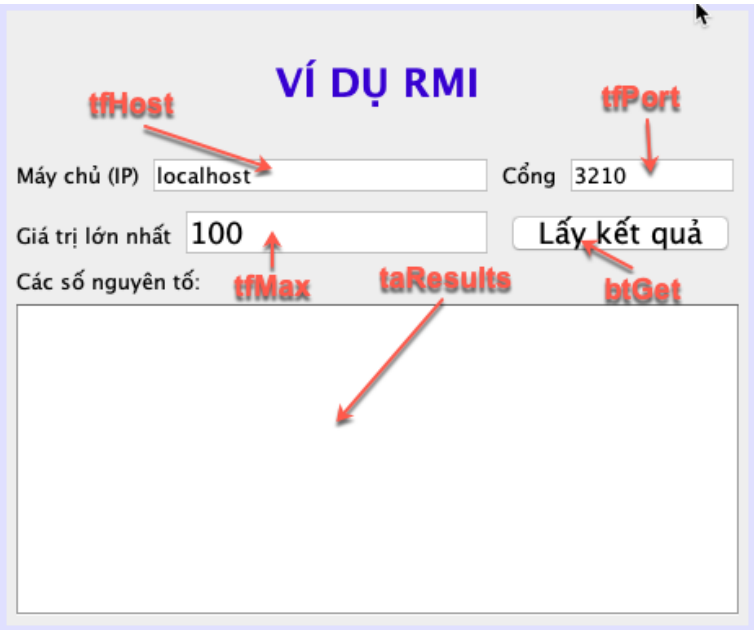
```
public class Prime extends UnicastRemoteObject implements PrimeInterface{
    //Constructor
    public Prime() throws RemoteException {
    }
    @Override
    public boolean isPrime(int x) throws RemoteException {
        for (int i = 2; i <= Math.sqrt(x); i++) {
            if (x%i == 0) {
                return false;
            }
        }
        return true;
    }
}
```

Bước 4: Lập trình cho Server. Trong package *RMI* tạo class đặt tên là *Server*. Chúng ta tạo một `Registry` trên cổng bất kỳ (chẳng hạn 3210) rồi ràng buộc (bind) một *PrimeService* cho một đối tượng thuộc lớp `Prime` trên đó.

```
public class Server {
    private final int PORT = 3210;
    public static void main(String[] args) {
        new Server().run();
    }
    public void run() {
        try {
            Registry reg = LocateRegistry.createRegistry(PORT);
            reg.rebind("PrimeService", new Prime());
            System.out.println("Máy chủ đang chạy...");
        } catch (RemoteException ex) {
```

```
            System.out.println("Không thể khởi chạy máy chủ!!!");
        }
    }
}
```

Bước 5: Lập trình cho Client. Trong project *RMI_Prime_Client* tạo một package đặt tên là *RMI*. Trong package này tạo một JFrame Form đặt tên là *Client*. Thiết kế giao diện cho *Client* như sau:



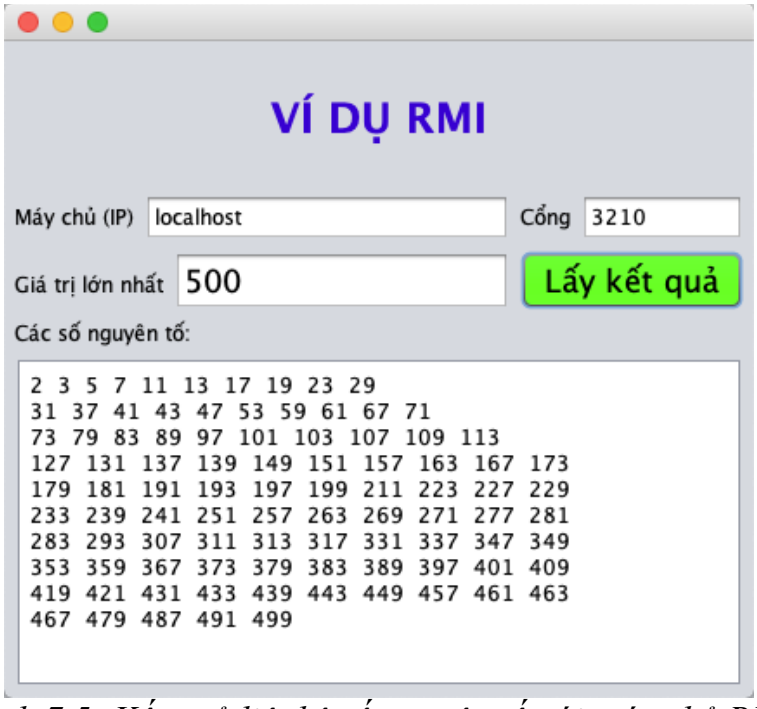
Hình 7.4: Thiết kế giao diện liệt kê số nguyên tố

Lập trình cho sự kiện người dùng nhấp chuột vào nút *Lấy kết quả* như sau:

```
private void btGetActionPerformed(java.awt.event.ActionEvent evt) {
    try {
        String host = tfHost.getText();
        int port = Integer.parseInt(tfPort.getText());
        int max = Integer.parseInt(tfMax.getText());
        taResults.setText("");
        Registry reg = LocateRegistry.getRegistry(host, port);
        NumberInterface prime =
(NumberInterface)reg.lookup("PrimeService");
        int count = 0;
        for (int i = 2; i <= max; i++) {
            if (prime.isPrime(i)) {
                taResults.append(i + "   ");
                count++;
            }
            if (count == 10) {
                taResults.append("\n");
                count = 0;
            }
        }
    } catch (RemoteException ex) {
        JOptionPane.showMessageDialog(null, "Không kết nối được tới máy
chủ!!!", "Lỗi", 0);
```

```
    } catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(null, "Giá trị lớn nhất phải là
số nguyên!!!", "Lỗi", 0);
    } catch (NotBoundException ex) {
        JOptionPane.showMessageDialog(null, "Không tìm thấy dịch
vụ!!!", "Lỗi", 0);
    }
}
```

Chạy *Server* trước, sau đó chạy *Client*. Trong form xuất hiện, nhập giá trị lớn nhất bất kỳ, giả sử là 500. Nhấn nút *Lấy kết quả*, chúng ta nhận được như hình bên dưới:



Hình 7.5: Kết quả liệt kê số nguyên tố với máy chủ RMI

***Ví dụ 7-2**. Viết chương trình tìm ước số chung lớn nhất và kiểm tra tính nguyên tố của hai số nguyên dương bất kỳ, sử dụng kỹ thuật lập trình RMI. Phương thức kiểm tra số nguyên tố và phương thức tính ước số chung lớn nhất của hai số được triệu gọi từ xa.*

Bước 1: Tạo 2 project *RMI_XuLySo_Client* và *RMI_XuLySo_Server*.

Bước 2: Trong project *RMI_XuLySo_Server* tạo một package đặt tên là *Core*. Trong package này tạo một interface đặt tên là `NumberInterface` với nội dung như sau:

```
public interface NumberInterface extends Remote{
    public int ucln(int a, int b) throws RemoteException;
    public boolean isPrime(int x) throws RemoteException;
}
```

Phương thức `ucln(int a, int b)` dùng để dùng để tìm ước số chung lớn nhất của hai số nguyên `a` và `b`.

Phương thức `isPrime(int x)` dùng để kiểm tra một số `x` có phải là số nguyên tố hay không.

Sao chép package *Core* sang project *RMI_XuLySo_Client*.

Bước 3: Hiện thực `NumberInterface`. Trong project *RMI_XuLySo_Server* tạo một package mới đặt tên là *RMI*. Trong package này tạo một lớp mới đặt tên là `NumberClass`.

Hiện thực hóa interface trong lớp này như sau:

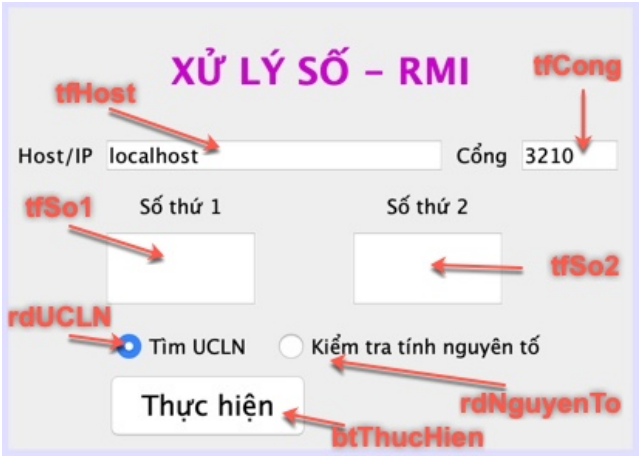
```
public class NumberClass extends UnicastRemoteObject implements
NumberInterface{
    //Constructor
    public NumberClass() throws RemoteException {
    }
    @Override
    public int ucln(int a, int b) throws RemoteException {
        while (a != b) {
            if (a>b) {
                a = a-b;
            } else {
                b = b-a;
            }
        }
        return a;
    }
    @Override
    public boolean isPrime(int x) throws RemoteException {
        for (int i = 2; i <= Math.sqrt(x); i++) {
            if (x%i == 0) {
                return false;
            }
        }
        return true;
    }
}
```

Bước 4: Lập trình cho Server. Trong package *RMI* tạo class đặt tên là *Server*. Chúng ta tạo một `Registry` trên cổng bất kỳ (chẳng hạn 3210) rồi ràng buộc (bind) một tên là *NumberService* cho một đối tượng thuộc lớp `NumberClass` trên đó.

```
public class Server {
    private final int PORT = 3210;
    public static void main(String[] args) {
        new Server().run();
    }
    public void run() {
        try {
```

```
            Registry reg = LocateRegistry.createRegistry(PORT);
            reg.rebind("NumberService", new NumberClass());
            System.out.println("Máy chủ đang chạy...");
        } catch (RemoteException ex) {
            System.out.println("Không thể khởi chạy máy chủ!!!");
        }
    }
}
```

Bước 5: Lập trình cho Client. Trong project *RMI_XuLySo_Client* tạo một package đặt tên là *RMI*. Trong package này tạo một JFrame Form đặt tên là *Client*. Thiết kế giao diện cho *Client* như sau:



Hình 7.6: Thiết kế giao diện xử lý số RMI

Lập trình cho sự kiện người dùng nhấp chuột vào nút *Thực hiện* như sau:

```
private void btThucHienActionPerformed(java.awt.event.ActionEvent evt) {
     try {
          String host = tfHost.getText();
          int port = Integer.parseInt(tfCong.getText());
          int n1 = Integer.parseInt(tfSo1.getText());
          int n2 = Integer.parseInt(tfSo2.getText());
          Registry reg = LocateRegistry.getRegistry(host, port);
          NumberInterface nService =
(NumberInterface)reg.lookup("NumberService");
          if (rdUCLN.isSelected()) {
               JOptionPane.showMessageDialog(null, "Ước chung lớn nhất là
" + nService.ucln(n1, n2), "Kết quả", 1);
          } else {
               if (nService.isNguyenTo(n1) && !nService.isNguyenTo(n2)) {
                    JOptionPane.showMessageDialog(null, "Số thứ nhất là số
nguyên tố.", "Kết quả", 1);
               }
               if (!nService.isNguyenTo(n1) && nService.isNguyenTo(n2)) {
                    JOptionPane.showMessageDialog(null, "Số thứ hai là số
nguyên tố.", "Kết quả", 1);
               }
               if (nService.isNguyenTo(n1) && nService.isNguyenTo(n2)) {
                    JOptionPane.showMessageDialog(null, "Cả hai đều là số
nguyên tố.", "Kết quả", 1);
```

```
                }
                if (!nService.isNguyenTo(n1) && !nService.isNguyenTo(n2))
{
                    JOptionPane.showMessageDialog(null, "Cả hai đều không
là số nguyên tố.", "Kết quả", 1);
                }
            }
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(null, "Dữ liệu không hợp lệ!!!",
"Lỗi", 0);
        } catch (RemoteException ex) {
            JOptionPane.showMessageDialog(null, "Không thể kết nối tới máy
chủ!!!", "Lỗi", 0);
        } catch (NotBoundException ex) {
            JOptionPane.showMessageDialog(null, "Không tìm thấy dịch
vụ!!!", "Lỗi", 0);
        }
}
```

Chạy *Server* trước, sau đó chạy *Client*. Trong form xuất hiện, nhập giá trị thích hợp. Nhấn nút *Thực hiện*, chúng ta nhận được như hình bên dưới:

Hình 7.7: Kết quả tìm ước chung lớn nhất của hai số

Hình 7.8: Kết quả kiểm tra tính nguyên tố của hai số

Ví dụ 7-3. *Viết chương trình xử lý xâu ký tự sử dụng kỹ thuật lập trình RMI. Các chức năng xử lý xâu gồm có: chuyển thành xâu in thường, chuyển thành xâu in hoa, chuẩn hóa xâu (xóa các dấu cách thừa), đếm số từ của xâu. Các phương thức xử lý xâu được triệu gọi từ xa.*

Bước 1: Tạo 2 project *RMI_XuLyXau_Client* và *RMI_XuLyXau_Server*.

Bước 2: Trong project *RMI_XuLyXau_Server* tạo một package đặt tên là *Core*. Trong package này tạo một interface đặt tên là `StringInterface` như sau:

```
public interface StringInterface extends Remote{
    public String lower(String st) throws RemoteException;
    public String upper(String st) throws RemoteException;
    public String standardize(String st) throws RemoteException;
    public int countWord(String st) throws RemoteException;
}
```

Sao chép package *Core* sang project *RMI_XuLyXau_Client*.

Bước 3: Hiện thực `StringInterface`. Trong project *RMI_XuLyXau_Server* tạo một package mới đặt tên là *RMI*. Trong package này tạo một lớp mới đặt tên là `StringClass`.

Hiện thực hóa interface trong lớp này như sau:

```
public class StringClass extends UnicastRemoteObject implements
StringInterface{
    //Constructor
    public StringClass() throws RemoteException {
    }
    @Override
    public String upper(String st) throws RemoteException {
        return st.toUpperCase();
    }
    @Override
    public String lower(String st) throws RemoteException {
        return st.toLowerCase();
    }
    @Override
    public String standardize(String st) throws RemoteException {
        String result = st.trim();
        while (result.contains("  ")) {
            result = result.replace("  ", " ");
        }
        return result;
    }
    @Override
    public int countWord(String st) throws RemoteException {
        String str = this.standardize(st);
        int count=0;
        if (str.equals("")) {
            return count;
        } else {
```

```
            count = 1;
        }
        for (int i = 0; i < str.length(); i++) {
            if (str.charAt(i) == ' ') {
                count++;
            }
        }
        return count;
    }
}
```

Bước 4: Lập trình cho Server. Trong package *RMI* tạo class đặt tên là *Server*. Chúng ta tạo một `Registry` trên cổng bất kỳ (chẳng hạn 3210) rồi ràng buộc (bind) một *StringService* cho một đối tượng thuộc lớp `StringClass` trên đó.

```
public class Server {
    private final int PORT = 3210;
    public static void main(String[] args) {
        new Server().run();
    }
    public void run() {
        try {
            Registry reg = LocateRegistry.createRegistry(PORT);
            reg.rebind("StringService", new StringClass());
            System.out.println("Máy chủ đang chạy...");
        } catch (RemoteException ex) {
            System.out.println("Không thể khởi chạy máy chủ!!!");
        }
    }
}
```

Bước 5: Lập trình cho Client. Trong project *RMI_XuLyXau_Client* tạo một package đặt tên là *RMI*. Trong package này tạo một JFrame Form đặt tên là *Client*. Thiết kế giao diện cho *Client* như sau:

Hình 7.9: Thiết kế form xử lý xâu theo kỹ thuật RMI

Lập trình cho sự kiện người dùng nhấp chuột vào nút *Xử lý* như sau:

```
private void btProcessActionPerformed(java.awt.event.ActionEvent evt) {
     String host = tfHost.getText();
     int port = Integer.parseInt(tfPort.getText());
     String st = tfInput.getText();
     try {
          Registry reg = LocateRegistry.getRegistry(host, port);
          StrInterface StrObj =
(StrInterface)reg.lookup("StringService");
```

```
            String result;
            switch (cbFunction.getSelectedIndex()) {
                 case 0:
                 result = StrObj.lower(st);
                 JOptionPane.showMessageDialog(null, result, "Kết quả", 1);
                 break;
                 case 1:
                 result = StrObj.upper(st);
                 JOptionPane.showMessageDialog(null, result, "Kết quả", 1);
                 break;
                 case 2:
                 result = StrObj.standardize(st);
                 JOptionPane.showMessageDialog(null, result, "Kết quả", 1);
                 break;
                 case 3:
                 result = StrObj.countWord(st)+"";
                 JOptionPane.showMessageDialog(null, result, "Kết quả", 1);
                 break;
            }
        } catch (RemoteException ex) {
            JOptionPane.showMessageDialog(null, "Không thể kết nối tới máy
chủ!", "Lỗi", 0);
        } catch (NotBoundException ex) {
            JOptionPane.showMessageDialog(null, "Không tìm thấy dịch vụ!",
"Lỗi", 0);

        }
}
```

Chạy *Server* trước, sau đó chạy *Client*. Trong form xuất hiện, nhập một xâu bất kỳ, chọn chức năng cần xử lý và nhấn nút *Xử lý*, chúng ta nhận được kết quả như hình bên dưới:

Hình 7.10: Kết quả chuyển xâu thành in hoa

Hình 7.11: Kết quả đếm số từ của xâu

CÂU HỎI, BÀI TẬP VẬN DỤNG:

1. Trình bày khái niệm về kỹ thuật lập trình RMI?
2. Trình bày quy trình tạo một ứng dụng mạng sử dụng kỹ thuật lập trình RMI?
3. Sử dụng kỹ thuật lập trình RMI, viết chương trình Java theo mô hình Client/Server xử lý xâu như sau:
   - Tính số từ của xâu.
   - Tìm xâu đảo ngược của xâu. Ví dụ: "ABCD" → "DCBA".
4. Sử dụng kỹ thuật lập trình RMI, viết chương trình Java theo mô hình Client/Server thực hiện tính toán như sau:
   - Chuyển một số sang hệ Hexa-Decimal (hệ thập lục phân).
   - Kiểm tra xem số đã cho có phải số hoàn hảo không?

   (N là số hoàn hảo nếu N có tổng các ước số bằng chính nó - trừ ước là N)
5. Sử dụng kỹ thuật lập trình RMI, viết chương trình Java theo mô hình Client/Server thực hiện tính toán như sau:
   - Tìm ước số chung lớn nhất của A và B.
   - Tìm xem có số nào trong hai số là số nguyên tố không?
6. Viết chương trình trò chơi TicTacToe giữa 2 người trên 2 máy tính khác nhau sử dụng kỹ thuật lập trình RMI?
7. Viết chương trình trò chơi đuổi hình bắt chữ bằng kỹ thuật lập trình RMI với hình ảnh và dữ liệu câu hỏi nằm trên máy chủ?
8. Viết chương trình thi trắc nghiệm với bộ câu hỏi nằm trong cơ sở dữ liệu trên máy chủ sử dụng kỹ thuật lập trình RMI?
9. Viết chương trình chuyển đổi mệnh giá các loại tiền tệ với tỉ giá được lưu trữ ở máy chủ, sử dụng kỹ thuật lập trình RMI?
10. Viết chương trình cung cấp thông tin thời tiết cho một số thành phố tại Việt Nam với dữ liệu thời tiết được lưu trữ ở máy chủ, sử dụng kỹ thuật lập trình RMI?

TÀI LIỆU THAM KHẢO

1. Elliotte Rusty Harold, *Java Network Programming, 4th Edition*, O'Reilly, USA, 2014.
2. Kenneth L. Calvert, Micheal J. Donahoo, *TCP/IP Sockets in Java: Practical guide for programmers, 2nd Edition*, Elsevier, USA, 2008.
3. Richard M. Reese, *Learning Network Programming with Java*, Packt, UK, 2015.
4. William Stallings, *Data and Computer Communications, 10th Edition*, Prentice Hall, USA, 2013.
5. James F. Kurose, Keith W. Ross, *Computer Networking: A Top-Down Approach, 7th Edition*, Addison-Wesley, USA, 2017.
6. Cay S. Horstmann, *Core Java, Volume I - Fundamentals, 10th Edition*, Prentice Hall, USA, 2016.
7. Cay S. Horstmann, *Core Java, Volume II - Advanced Features, 10th Edition*, Prentice Hall, USA, 2017.